A MODEL FOR

ERROR DETECTION AND CORRECTION

Pavel Brazdil

Ph.D.

University of Edinburgh

1981

I declare that this thesis has been composed by myself and that the work described in this thesis is a work of my own.

Acknowledgements

I wish to thank Dr Gordon Plotkin who has supervised my work for his interest in my work and many helpful discussions, and also for comments on the draft of this thesis. I also thank to Prof. B Meltzer for his help during the initial period of my studies in Edinburgh.

I further wish to thank everybody in the Department of Artiticial Intelligence in Edinburgh who helped me in various ways.

My thanks also to IBM UK Laboratories Ltd. for the financial support during the first two years of my stay in Edinburgh, and for their understanding which enabled me to complete this thesis. Also, I thank IBM UK Scientific Centre for their helpful attitude which aided me in carrying out some of this work on a part time basis.

ABSTRACT

The aim of our work is to investigate how a relatively small set of clauses can be transformed into a running program capable of solving a number of problems. The problems are chosen from the domain of simple arithmetic, algebra and letter series completion. We describe how the problems are solved, how errors are detected and corrected by modification of the existing clauses.

Various techniques useful in the process of error detection and correction are described in detail. Two types of errors are dealt with: selection errors arising due to incorrect selection of clauses, and instantiation errors arising when the partial results (subgoals) are not specific enough.

The system described was implemented in Prolog.

CONTENTS

1 INTRODUCTION 4
1.1 Main Objectives of Our Work 4
1.2 Basic Assumptions 5
1.3 Overview of Our System 10
1.4 What Our System Can Do 20
1.5 Implementation 41

2 SEARCH AND DETECTION OF ERRORS 42
2.1 Introduction 42
2.2 Clause Selection 44
2.3 Clause Application 48
2.4 Detection of Selection Errors 49
2.5 Reselection and Backtracking 54
2.6 Detection of Instantiation Errors 55
2.7 Summary 57

3 CORRECTION OF SELECTION ERRORS 59
3.1 Introduction 59
3.2 Objectives of Error Correction 65
3.3 Correction of Simple Selection Errors 66
3.4 Correction of Conflicting Selection Errors 70
3.5 Decomposition of Contexts 78
3.6 How General Constraints are Generated 81
3.7 Modification of Existing Constraints 85
3.8 Dealing with Multiple Conflicts 88
3.9 Reorganization of Priority Orderings 92
3.10 Preventing Recurrence of Errors 95
3.11 Learning from Examples 100
3.12 Dealing with Different Domains 105
3.13 Implementation 107
3.14 Experimental Results 108
3.15 Discussion 136

4 CORRECTION OF INSTANTIATION ERRORS 142
4.1 Introduction 142
4.2 Detection of Instantiation Errors 144
4.3 Correction of Instantiation Errors 149
4.4 Elimination of Additional Selection Errors . . . 154
4.5 Some Problems with Variables 155
4.6 Implementation 158
4.7 Experimental Results 160
4.8 Summary 171

5 EXTENDED SYSTEM ELM2 173
5.1 Introduction 173
5.2 Goal Stack (GS) Clauses 176
5.3 Generation of GS Constraints 177
5.4 Reordering Goals with GS Clauses 179
5.5 Implementation 180
5.6 Experimental Results 181
5.7 Summary 201

6 RELATION TO OTHER WORK 203
6.1 Winston: Learning Structural Descriptions . . . 203
6.2 Biermann: Inference of Programs from Examples . 211
6.3 Waterman: Adaptive Production Systems 224

7 SUMMARY AND CONCLUSIONS 232
7.1 Problems of Execution Control 232
7.2 Modifications without Side-effects 238
7.3 Choice of Alternatives 240
7.4 Preventing Recurrence of Errors 241
7.5 Interactivity 243
7.6 Applications 244

REFERENCES 246
APPENDIX - Glossary 248

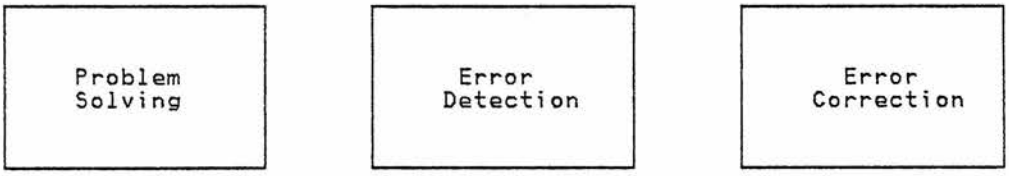
1 INTRODUCTION

1.1 MAIN OBJECTIVES OF OUR WORK

This thesis describes our work in the area of program development. Our main objective was to see how a given simple program can be transformed so that rather a wide variety of problems will be solved.

We believe that this is an important subject to investigate. Any understanding gained can help us to improve the methodology of programming in general. Also, it can enhance our understanding about learning, since program development and learning are closely related.

We believe that problem solving and clause modification are closely related. Generally, it is desirable to observe each solution, see what is wrong, and try to modify the existing program to improve the situation. The three main processes involved are:



We followed the approach of AI in our investigation: In order to find more about these processes we decided to build a 'model'.

The design and thus also our investigation was affected by the following questions:

- What kind of problem solving system should we choose in our investigation?
- Should problems be given?
- What criteria should we use for error detection?
- What type of errors should we deal with?
- What type of information should be used for error correction?
- What is the extent of the intended modifications?

The answers to these questions affected the line of the investigation we have taken. In the following section we shall present our answers to these questions.

1.2 BASIC ASSUMPTIONS

Problem Solving System

Investigation into error detection and modification could not be performed without a problem solving system. One has to have a problem solving system first. A question arises whether all systems are equally good, or whether some are better than others.

We came to a conclusion that a simpler system is better than a more sophisticated one. One reason for this is that a sophisticated system makes fewer mistakes than a simpler one (by definition), and so there is less material to investigate.

Also, it is often difficult to see why each error arose. To detect errors in a sophisticated system one needs also a sophisticated error detection system, and both take time to develop.

This is exactly why we did not use a relatively complex 'hierarchical planning system' which we were intending to develop (*).

To save ourselves the work of designing a completely new system we decided to use an existing system instead. Prolog (Warren, 1977) seemed like a good candidate. It was developed as a result of investigations into the use of logic in problem solving, and the work in theorem proving (Kowalski, 1979).

We have observed that it is not always easy to modify the existing Prolog programs and this is why we developed our own version of Prolog. Our programs, too, consist of 'clauses' (**), which are similar to the clauses used in Prolog. However, the clauses which we use have a somewhat different syntax and semantics which makes it easier for the system to perform all the modifications needed.

Choice of Problems

We decided that our investigation should be based on how concrete problems are solved. The problems to be solved are

---

(*) The system was described in our research proposal in 1975 (unpublished). A similar system to the one we wanted to develop was described by Solomon (1976).

(**) In many places we use the terminology of Kowalski (1979), but some of the terms have more specific meaning. The meaning of various terms used is given in the Glossary.

given to the system. This is by no means the only way programs could be developed.

What we could do is examine the program in the abstract. That is, we could assume that some clause has been selected and then examine how its subgoals could be solved. We would use a 'symbolic interpreter' to identify various shortcoming of programs.

Burstall and Darlington (1975), for example, have shown how redundant computations can be identified and eliminated. Symbolic execution plays a significant role in their system.

Criteria for Error Detection

Our basic assumption was that errors should be detected by examining how problems are being solved. The question is - which criteria should be used for detection of errors ?

There seem to be two possible lines of approach. One is that to use an independent source of information showing how problems should be solved. Another possibility is to use one's own criteria of how problems should be solved. We may notice, for example, that the solution contains a number of unnecessary steps which could be deleted.

We have adopted first of the two methods. An information is given to the system showing how problems are to be solved. It is provided in the form of goal traces.

The conditions for detection and correction of errors have been idealized in some ways. For example, the same problem is always solved in the same way. In real life the information provided may be different from one occasion to another. This is because the people involved often have very different

background. They may also have different objectives.

The amount of information given often depends on how much each individual student knows and what his progress is. In our system the information given is always constant.

It would be interesting to examine various interactions between a teacher and its pupil and see how these interactions could be represented in our model.

Further, it would be interesting to follow up the second line of approach mentioned, and investigate how the system could be more independent and detect errors on its own. Two different aims could be followed: first, how to obtain solutions quicker; second, how to detect that various solutions which might be obtained are inconsistent with one another.

The system could try to see, for example, if various steps are repeated and then try to transform the original program so that this is avoided. Such program transformations have been performed by Darlington and others (Burstall & Darlington, 1975). The system could also try to identify various steps which are often repeated and then introduce 'macros' or 'macro-operators', as, for example, Fikes has suggested (1972).

To detect whether the solutions are inconsistent the system could do the following. It could try to solve the given problem in different ways and then check if the solutions obtained are the same. If, for example, program P' was a more efficient version of some program P, the system could check if both programs give the same result. If they did not the system could try to find out in which step the error was made and then correct the program accordingly.

Type of Errors

We decided to deal with the following two types of errors: selection errors and instantiation errors.

Selection errors arise if all clauses to be used during the solution of given problems are given, but some of the conditions affecting their selection are incorrect (or missing). Consequently, a wrong clause is selected and a wrong result is obtained.

Instantiation errors arise if the results obtained are not specific enough. Typically, the errors arise if a variable is given as an answer instead of a specific constant. Both types of errors are discussed later in more detail (chapter 2.4).

We realized that our ideas on the type of errors we wanted to detect were affected by what techniques we would use to detect them. Sussman (1975), for example, classifies errors differently from us, because he uses a different method of detecting them.

Extent of Modifications Required

All errors which we examined in detail were corrected by the modification of one clause. We realize that we have examined a certain type of errors which are easier to correct.

Of course, there are errors which can be corrected by modifications of several clauses only. Such errors arise, for example, if some predicate used in several clauses is to be replaced by another one.

Modifications are often easier if a suitable language is used. In chapter 5 we discuss an extended version of our

system. We show that some errors are more easily corrected in it than in our original system. This is because the clauses are written differently - in a form which is more convenient for error correction.

1.3 OVERVIEW OF OUR SYSTEM

The processes of problem solving, error detection and correction are very complex indeed. To be able to study these processes better, we have simulated various phases involved. Our model was implemented in Prolog on a DEC-10. Our model will be referred to as the Experimental Learning Model, or as ELM1. An extended version of our system which is described in chapter 5 will be referred to as ELM2. Let us now see what the main parts of our system are.

1.3.1 Problem Solving Subsystem

The aim of the problem solving subsystem is to solve the given problems. The existing clauses are used in the process. These include a certain number of clauses given initially, and possibly some new clauses generated by the system. Clauses in our system are written as follows:

```
G <- Cs & ! & Rs,
```

where

G represents the 'clause head' consisting of a predicate,

Cs represents particular predicates called 'constraints',

! is a special symbol used to separate Cs and Rs,

Rs represents predicates in the 'clause body'.

Clauses do not need to contain any constraints. Examples of such clauses are given in the following.

```
asoc:  X1+(X2+X3)=X4   <-   !  & (X1+X2)+X3=X4
subs:  X1+X2=X3        <-   !  & X1=X4  & X4+X2=X3
```

Clause 'asoc', for example, expresses associativity of '+'. This clause says that goal 'X1+(X2+X3)=X4' should be solved by solving the subgoal '(X1+X2)+X3=X4'. Clause 'subs' says that goal 'X1+X2=X3' should be solved by solving the subgoals 'X1=X4 & X4+X2=X3'. The equation 'X1=X4' deals with a particular subterm from the original equation.

The search for a solution is quite straightforward. For each given goal a number of clauses can be selected from among the existing clauses. All clauses selected are then applied. The search proceeds in parallel on different branches. After some branch has been extended by application of one clause, other branches are extended to the same depth, as in the breadth-first search.

The following figure shows how search trees are normally expanded. The current goal is represented by Gj'. The goal Gj' can be solved in two different ways (using two different clauses). The subgoals Gk" or Gk"' represent the new subgoals obtained in each case.

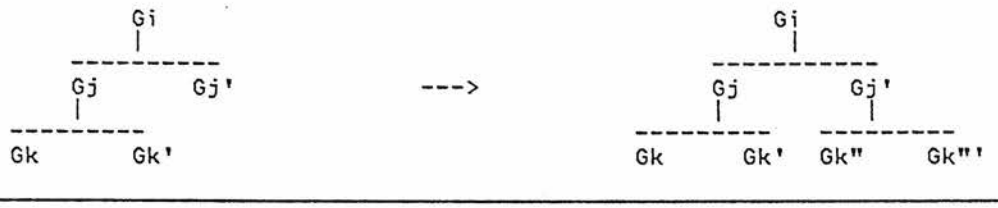


Fig. 1.1 Expansion of Search Trees

All the search trees shown in the following are, in fact, OR-trees, just like the tree just shown. The number of branches in each node will, of course, depend on how many clauses have been selected there.

Several conditions affect selection of clauses. Each clause can be selected only if it matches the current goal, as in Prolog (Warren, 1977). However, other conditions are also tested. Each clause can be selected only if the constraints are 'true'. Selection of clauses is also affected by priority orderings among the existing clauses.

Selection of clauses is also affected by certain types of errors. Certain errors indicate that the solution cannot be reached by continuing the search on any of the existing branches. If such errors are detected reselection is performed. During reselection the clauses which would have been normally selected with the particular goal are ignored. Selection is continued with the remaining clauses only. If errors are detected again reselection is repeated. If all clauses have been considered and rejected backtracking occurs. The system uses one of the preceding nodes in the search tree and performs reselection for that node. Backtracking may be repeated, too.

Application of each clause selected creates a new node in the search tree. Each node contains a list of subgoals to be solved. New subgoals introduced are dealt with last.

The problem solving subsystem is described in chapter 2 in more detail.

1.3.2 Error Detection Subsystem

Error detection is performed on the basis of comparison of the information showing how problems are solved, and the information showing how problems should be solved. This information is given to the system. It is provided in the form of goal traces.

A given goal trace is a sequence of goals (or subgoals) which would have been produced by application of clauses in an error free system.

Example

If, for example, goal '3+(1+1)=X1' was supposed to be transformed into '(3+1)+1=X1' on the basis of associativity, the goal trace would contain the following goals:

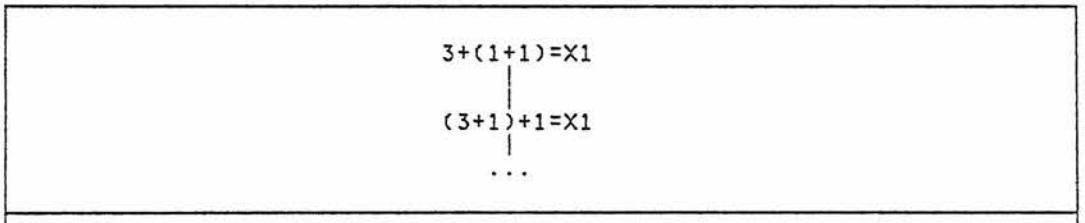


Fig. 1.2 An Example of a Goal Trace

Error detection is performed in the following way. The goals obtained by the system are compared with the corresponding

goals in the goal trace. Errors are detected if the goals differ.

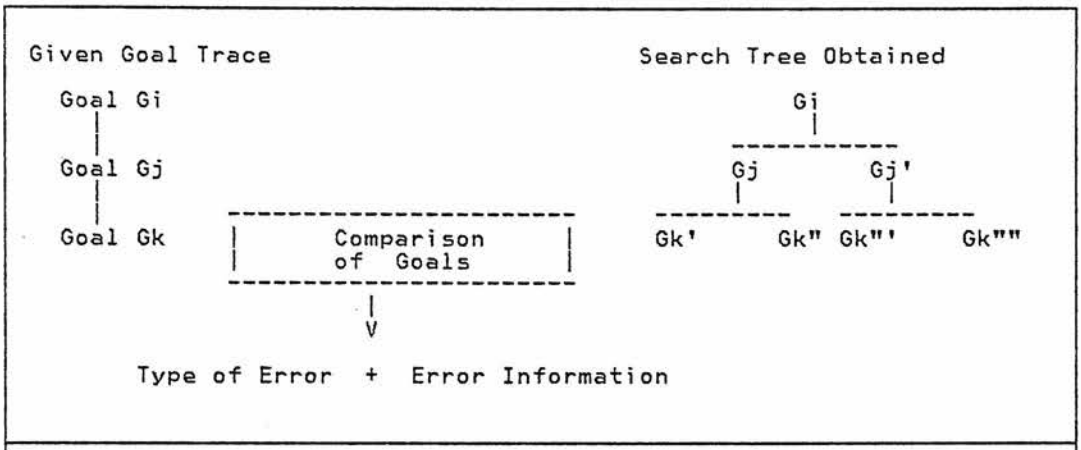


Fig. 1.3 Detection of Errors

The following types of errors may be detected in our system:

- selection errors,
- instantiation errors.

Selection errors are detected if the goals obtained by the system do not match the corresponding goals in the goal trace. If there is at least one branch in the tree where the goals are the same as the goals in the trace the selection errors is classified as a 'simple selection error'. If there is no such branch the error is a 'conflicting selection error'.

If all the goals obtained by the system differ from the goals in the goal trace the chances of finding the solution by continuing the search are small. This is why the search is not continued any further and a reselection (or backtracking) is performed.

We see that a reselection occurs after a conflicting selection error has been detected.

Instantiation errors are detected if the goals obtained by the system are not instantiated as they should be. That is if they contain some variables instead of constants (or terms).

Clearly, a variable is not acceptable as an answer if the problem is to calculate the sum of two integers, for example. The answer is not specific enough. We have assumed that if the intermediate subgoals are not specific enough it is unlikely that the solution would be. This is why instantiation errors are detected if the goals obtained by the system are not instantiated as they should be. More details about how errors are detected are given in chapter 2.4.

Error Information Stored

Detection or various types of errors is accompanied by storage of various pieces of information. This information is passed to the error correction subsystem later.

In our system different information is stored depending on which error has been detected.

With selection errors the error information includes the name of the clause which should have been selected with some particular goal(s). The goals are stored, too. The goals represent a context in which the clause mentioned should have been selected. This context is sometimes referred to as the selection context in the following.

With instantiation errors the error information includes the name of the faulty clause which needs to be modified and two clause instances (the faulty and the desired clause instance). In chapter 4.2 we explain how the two clause instances are obtained.

1.3.3 Error Correction Subsystem

No errors are corrected while the system searches for a solution. This is because the system has not yet had a chance to establish which clauses should be selected in each step. All errors are corrected after the search has terminated. The error information stored is used in the process.

In the following we shall describe how various types of errors are corrected by the system.

Correction of Simple Selection Errors

Simple selection errors arise if a number of clauses are selected instead of just one. They are corrected by the addition of new 'priority orderings'. If, for example, clauses Ci and Cj have been selected instead of just clause Ci, priority ordering Ci > Cj is generated. The system knows which clause should be given priority; the name of the clause has been identified when the error was detected.

Generation of each new priority ordering is accompanied by the storage of one particular 'context'. It is the context in which the particular error was detected. The contexts are used in correction of conflicting selection errors.

Correction of simple selection errors is discussed in detail in chapter 3.3.

Correction of Conflicting Selection Errors

Conflicting selection errors cannot be corrected by addition of priority orderings only. If they were, a conflicting system of priority orderings would have been obtained. The method used for correction of conflicting selection errors is referred to as conflict resolution. The method is described in detail in chapter 3.

If the ordering Ci > Cj already exists no error can be corrected by the introduction of Cj > Ci. Such conflicting priority orderings are never introduced by the system. Any error which would require the addition of such an ordering is corrected by the introduction of Cj'> Ci. Clause Cj' is a new version of clause Cj which is generated by the system. It contains new constraints.

The constraints of clause Cj' are generated so that selection of clause Ci would not be affected by the fact that that the ordering Cj' > Ci has been introduced. That is it should be possible to select clause Ci in its selection context. This context is referred to here as the rejection context of clause Cj'. Selection of clause Cj' in its own selection context should not be affected, however. This is the context in which the error was detected.

So, new constraints are generated on the basis of analysis of those two types of contexts. Various predicates to be used as constraints are tried out with various subterms of these contexts.

Some predicate may be used as a constraint if it is true when it is used with the selection context subterms. Moreover, the predicate must be false when it is used with the corresponding rejection context subterms. Chapter 3.4 explains the process of constraint generation in detail.

The aim of the system is to find constraints which are as general as possible. More general constraints are preferred since this minimizes the chances of generating 'overconstrained' clauses which could not be selected in any context.

The process of constraint generation does not stop after one constraint has been found. Indeed, all possible predicates satisfying the conditions mentioned are used. They are used as disjuncts in the final expression generated. The disjunctions of constraints are preferred because they are more general than any of the disjuncts.

The constraints which may be introduced by our system may contain one or two variables only.

Correction of Instantiation Errors

Instantiation errors are corrected differently from selection errors. The aim of the modifications is to ensure that variables in the 'faulty clause' are instantiated as appropriate.

The desired clause instance stored when the particular error was detected shows how this clause should have been instantiated. The faulty clause instance shows how the clause dealt with was instantiated at the time the error was detected. Both clause instances are analyzed by the system.

Various predicates from a given set are tried out to see if the 'faulty clause instance' can be instantiated as required. The new predicates found are referred to as variable instantiating predicates. After all such predicates have been found the faulty clause is modified. The method described is discussed in detail in chapter 4.3.

1.3.4 The Extended System ELM2

In the system just described (ELM1), only one goal plays a role in clause selection. Sometimes, however, it is necessary to consider other goals apart from the current goal, when deciding which clause should be selected. This is possible in our extended system (ELM2). The extended system is described in detail in chapter 5.

The clauses used in our extended system are different from the clauses used in ELM1. The main difference is that several predicates may appear in the clause head. An example of such a clause is given in the following:

```
asoc:  X1+(X2+X3)=X4 & G1 <-   !  & (X1+X2)+X3=X4
```

Each clause can be selected only if the clause head matches the current goals (the current goal stack). The constraints of these clauses may refer to any predicate in the clause head. As selection of clauses is affected by which goals appear on the goal stack, the new type of clauses are referred to as goal stack clauses (GS clauses).

The method for correcting conflicting selection errors has already been described. With GS clauses the selection and rejection contexts include the whole goal stack, and in this way the extended system ELM2 differs from the system ELM1 described before.

The extended system ELM2 is more powerful than ELM1. It is possible to teach it to do integer division, for example. After several problems have been solved ELM2 can solve all future problems without errors.

The system ELM1 cannot learn to do division with a similar set of clauses. That is ELM1 makes an attempt to correct each error, but the attempt is unsuccessful and so the errors keep recurring.

1.4 WHAT OUR SYSTEM CAN DO

In the following we shall show how our system learns to add integers, solve simple equations and predict the next letter in a given series. Our extended system can also learn to divide integers.

1.4.1 Learning to Add Integers

Our system can learn to add integers. It can learn to do simple additions like 3+2 or 4+3, but also more complex additions like ((3+1)+(3+1))+2, for example. The problems given to the system were written in the following way:

```
          3 + 2 =X1
          4 + 3 =X1
    (1+2)+(2+1)=X1
((3+1)+(3+1)+2 =X1
```

In the following we shall show how the ability to add integers is acquired by modification of the given set of clauses. The clauses given are shown in the following figure.

Clause 'asoc', for example, expresses associativity of '+'. Clause

'subs' says that the goal 'X1+X2=X3' should be decomposed into two subgoals : 'X1=X4' and 'X4+X2=X3'. Clause 'subz' is similar.

Clauses 'suc' and 'pred' specify successors and predecessors of integers. Predicate 'T=:X1' which is used in these clauses instantiates the variable 'X1' to whatever 'T' is.

No priority orderings were specified for the clauses shown.

```
asoc:  X1+(X2+X3)=X4   <-   !  & (X1+X2)+X3=X4
subs:  X1+X2=X3        <-   !  & X1=X4   & X4+X2=X3
subz:  X1+X2=X3        <-   !  & X2=X4   & X1+X4=X3
eq:    X1=X1           <-   !

pred:  2 =X1  <-  1+1=:X1 & !        suc:  1+1 =X1  <-  2=:X1 & !
       3 =X1  <-  1+2=:X1 & !              2+1 =X1  <-  3=:X1 & !
       etc.                               etc.
```

Fig. 1.4 The Initial Set of Clauses Given

These clauses will be modified by the system. The new clauses will enable the system to add integers without errors.

In the following we shall show how the problem '3+2=X1' was solved by the system. This problem was given to the system as a 'goal' to solve. This goal was accompanied by the goal 'write(X1)' to enable us and the system to see whether the result obtained was correct.

Figure 1.5 shows how the given problem was solved. Each branch in the tree shown represents a particular way of solving the current goals (or subgoals). The goal '3+2=X1', for example, can be solved in three different ways, using clause 'subz', or clause 'subs', or clause 'eq'. Each clause mentioned matches the goal '3+2=X1'. When, for example, clause 'subz' is used, the predicates in the clause body get instantiated into

2=X4 & 3+X4=X1.

Both predicates are used as new subgoals later. The subgoal '2=X4' is shown in our search tree, below clause 'subz'.

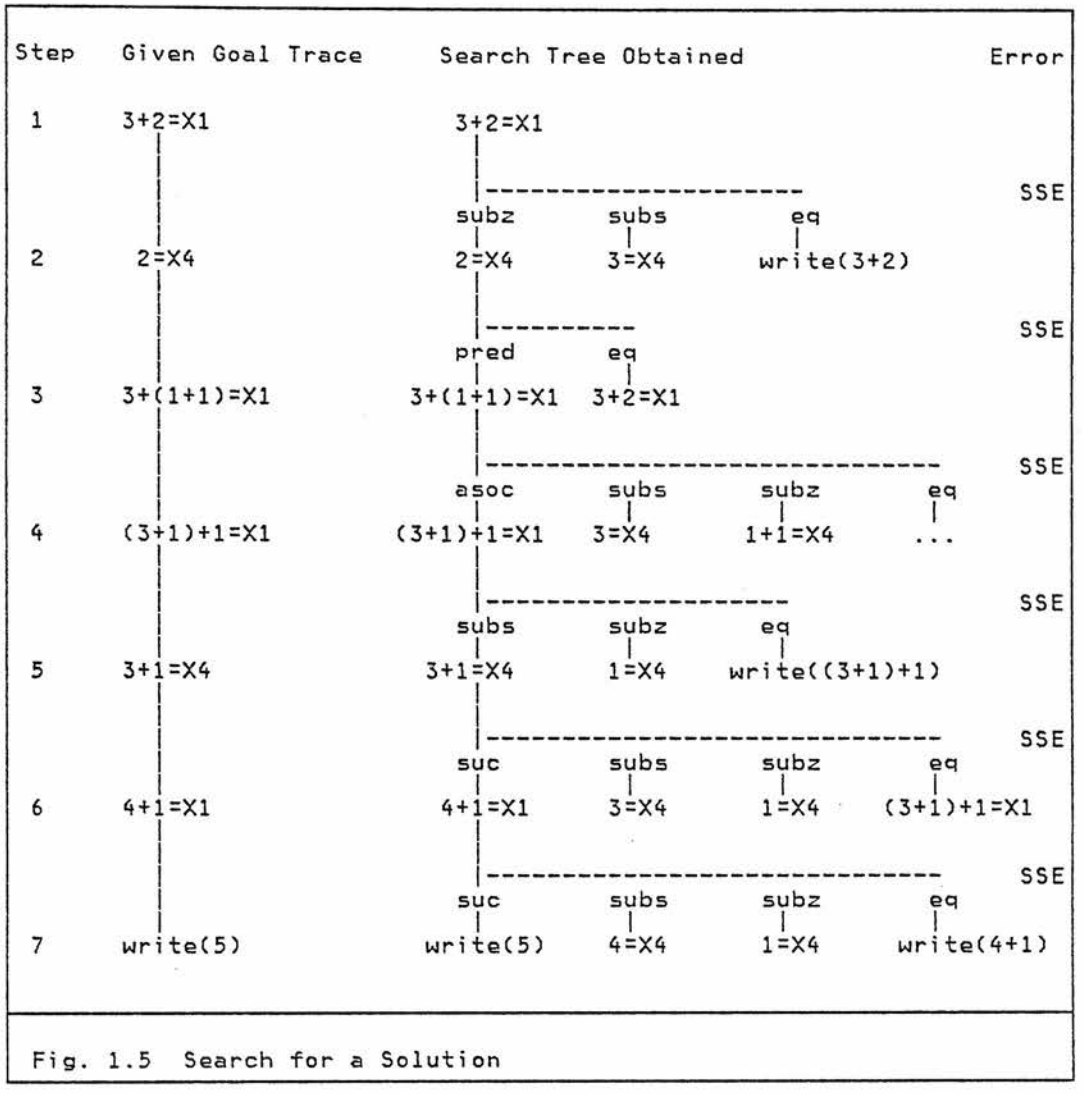


Fig. 1.5 Search for a Solution

Error Detection

Errors are detected by the system while the search tree is expanded. The goals obtained by the system are compared with the corresponding goals in the given goal trace. Errors are detected if the goals differ.

Let us see how one particular error is detected in our example. Let us

examine the search tree after step 1. The goal '2=X4' obtained by application of clause 'subz' matches the corresponding goal in the given goal trace. This is why the search can continue on this branch. The goal '3=X4' obtained by application of clause 'subs' does not match this goal, however. A selection error is detected as a result. The search is terminated on this branch. As search can continue on at least one of the branches the error is classified as a 'simple selection error' (SSE).

The following 'error information' is stored:

Current Goal:	Clause to be Selected:	Type of Error:
3+2=X1	subz	SSE

Detection of errors continues like this during the search. After the search has terminated, the system tries to correct the errors detected. The error information stored is used in the process.

Error Correction

The error detected in step 1 is corrected by introduction of two new priority orderings:

```
subz > subs,
subz > eq.
```

Each priority ordering specifies that when selection is performed priority should be given to clause 'subz' over the other clause mentioned.

Other errors detected are corrected in a similar way. The error in step 4, however, cannot be corrected by introduction of priority orderings only. If it was, a conflicting system of priority orderings would have been obtained:

```
subz > subs
subs > subz.
```

The error detected in step 4 is re-classified as a 'conflicting selection error'. This error is corrected by conflict resolution. Clause 'subs1' is introduced and given priority over the clause 'subz'. Clause 'subs1' is a new version of clause 'subs' containing new constraints. This is what the new clause looks like:

```
subs1: X1+X2=X3 <-  (X1=:(X5+X6)  v X2=:1)  & !  & X1=X4  & X4+X2=X3
```

The new constraints added appear before the special symbol '!'. We see that a disjunction of two new constraints has been added to the original clause. Predicate 'X1=:(X5+X6) specifies that this clause can be selected if variable 'X1' is instantiated into 'X5+X6' after the match of the clause head against the current goal. Predicate 'X2=:1' requires that variable 'X2' is instantiated to '1'.

A set of predicates to be used as constraints was given to the system beforehand. In this series of experiments the set included the following predicates:

int(Xi) true if Xi is an integer,
var(Xi) true if Xi is a variable and
Xi=:Xj which is described below.

The predicate Xi=:Xj is true if Xi and Xj are identical, or if they can be made identical by instantiating the variables in Xj.

After all errors have been corrected the system stops and waits for another problem. The priority orderings obtained are shown in the following figure.

```
                asoc  > subz > subs
          suc > subs1 > subz >  eq
                        pred >  eq
```

Fig. 1.6 Priority Orderings Obtained

We can verify that all errors have, in fact, been eliminated by our modifications. Only one clause is selected in step 1, for example. It is the clause 'subz' and this is right. Clause 'subs1' is not selected even though this clause has higher priority than clause 'subz'. Selection of clause 'subs1' is prevented by its constraints. This is not by chance. The constraints of 'subs1' have been generated so that selection of this clause would be prevented in this context (*).

The error in step 4, for example, has also been eliminated. The only clause selected in this step is clause 'subs1'. Selection of clause 'subz' and 'eq' is prevented by the priority orderings.

Other Problems of Addition

The problem '4+3=X1' which we gave to the system next was solved without errors. Even though no modifications had to be performed to eliminate errors, the clause 'subs1' generated before was modified again. It was modified using the method of 'learning from examples' described in chapter 3.11. According to this method, clauses can be modified by simplifying the existing disjunctions of constraints if certain conditions are satisfied. In our case clause 'subs1' was modified as follows:

(*) This context is the 'rejection context' of clause 'subs1'. The use of this context in the process of constraint generation is explained in chapter 3.4.

```
subs1:  X1+X2=X3  <-  X1=:(X5+X6)   & !  & X1=X4  & X4+X2=X3
```

In section 3.11 we explain how such modifications help to prevent future errors.

One simple selection error occurred in the course of the solution of the problem '(1+2)+(2+1)=X1' which was given to the system next. This error was corrected by introduction of one priority ordering (subs1 > asoc).

The problem '((3+1)+(3+1))+2=X1' was solved without errors. We believe that the system can now solve many similar problems without errors. The system can add any number of integers provided all the necessary 'successors' and 'predecessors' of the integers dealt with are given (eg. that '11' is a successor of '10').

We believe that our system could learn to do multi-column additions as well. It would be interesting to see which clauses one would need for that and how many new clauses would be generated.

Our system can learn to subtract integers in a similar way as it can learn to add. More details about how subtraction was learnt can be found in chapter 3.14.

1.4.2 Learning to Solve Simple Equations

Our system can also learn to solve simple equations like these:

(X1+3)+1=7	(2+X1)+(3+1)=7
(X1+4)+1=8	((2+1)+2)+X1=9
(2+1)+X1=5	((3+X1)+2)+1=8
(1+X1)+3=7	

The system of clauses acquired in the course of learning addition and subtraction were used in our experiments with simple equation solving. Three other clauses were added to them:

```
asod:  (X1+X2)+X3=X4   <-  !  &  X1+(X2+X3)=X4
com:    X1+X2=X3       <-  !  &  X2+X1=X3
over:   X1+X2=X3       <-  !  &  X1=X3-X2
```

Fig. 1.7 Additional Clauses Used in Simple Equation Solving

The additional clauses were given lower priority than the clauses used before, when addition and subtraction was being taught. If this was not done errors would arise with the problems of addition shown before. The following priority orderings are required to prevent that:

```
subs1 > asod
subz  >  com
subz  >  over
```

Fig. 1.8 Priority Orderings Added

Search and Detection of Errors

Fig. 1.9 shows how the equation '(X1+3)+1=7' is transformed into a problem of subtraction. This problem is solved without further difficulties. We can see how the goal trace given helps the system to find the solution.

In the first step, for example, clause 'subs1' is selected first. However, this is the wrong clause to use in this step. The goal trace given enables the system to discover the error. We see that goal 'X1+3=X4' obtained by application of clause 'subs1' does not match goal 'X1+(3+1)=7 in the goal trace. As there is no other branch in the tree where the search can continue, the error is classified as a 'conflicting selection error'. To continue the system performs reselection. Clause

'subs1' selected before is ignored and clauses 'asod' and 'subz' are selected as a result. Clause 'asod' is the correct clause to use.

Reselection is performed similarly twice later, in steps 2 and 4.

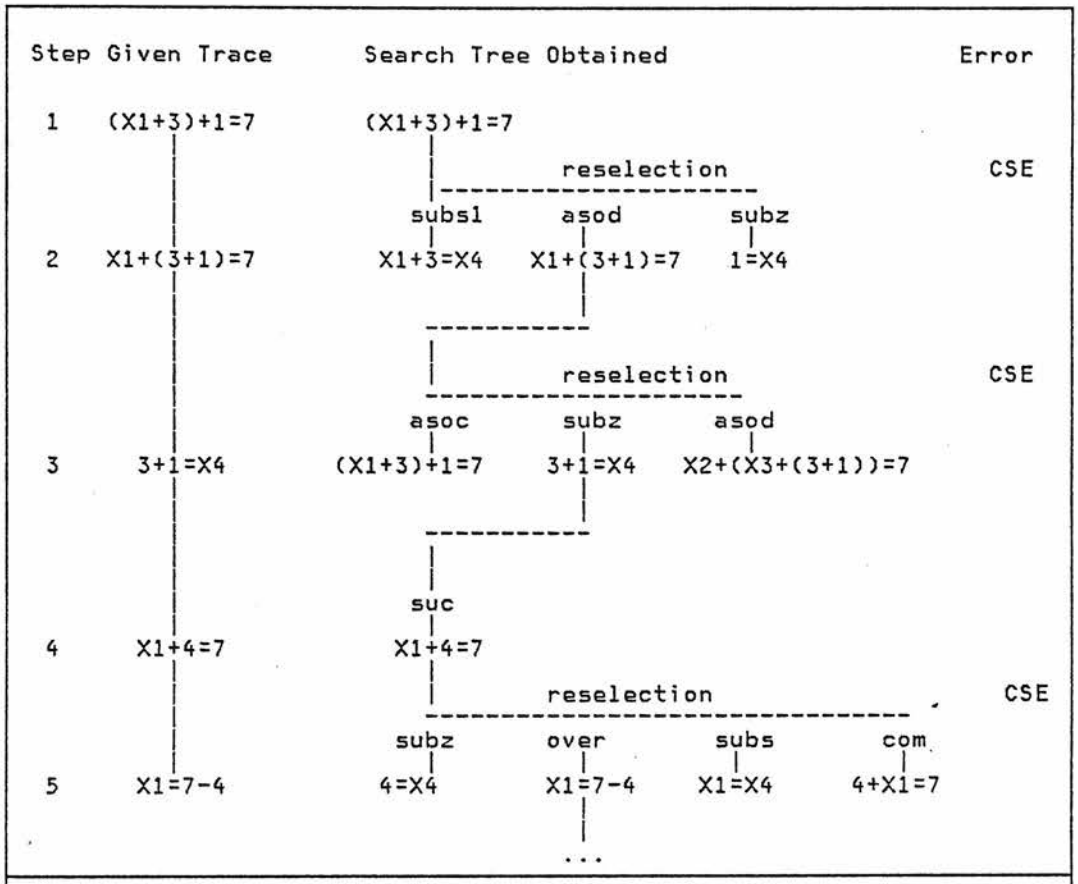


Fig. 1.9 Search for the Solution of (X1+3)+1=7

Correction of Errors

Each conflicting selection error detected is corrected using the method of conflict resolution, described in chapter 3.4. Correction of each error results in a generation of one new clause.

The error detected in step 1, for example, is corrected by generation of clause 'asod1'. This clause is given priority over clause 'subs1'. Clauses 'subz1' and 'over1' are generated to correct the errors detected in steps 2 and 4. The new clauses are shown in the following figure.

```
asod1:  (X1+X2)+X3 =X4   <-    (var(X1)  v X2=:3  v int(X4))  & ! &
                                X1+(X2+X3) =X4

subz1:  X1+X2=X3         <-    (var(X1)  v X2=:(3+X5))  & ! &
                                X2=X4     & X1+X4=X3

over1:  X1+X2=X3         <-     X2=:4     & !  &  X1=X3-X2
```

Fig. 1.10 New Clauses Generated

These are the priority orderings that were obtained:

```
                subz1 >          asoc  > subz >  subs
                 suc  > subs1 > asoc  > subz >  eq
                                      pred >  eq

      over1 > subz1 > asod1 > subs1          >  asod
                                      subz >  com
                                      subz >  over
```

Fig. 1.11 Priority Orderings Obtained

It is easy to verify that the previous equation can be solved without errors now. In step 1, for example, only clause 'asod1' is selected. Clauses 'over1' and 'subz1' are not selected even though they have higher priority than clause 'asod1'. Their selection is prevented by their constraints.

We notice that clause 'over1' in not constrained very well. This

clause can be selected only if 'X2' is 4. Because of this an error occurs with the equation '(X1+4)+1=8' which we gave to the system next. The following figure shows how this equation was solved.

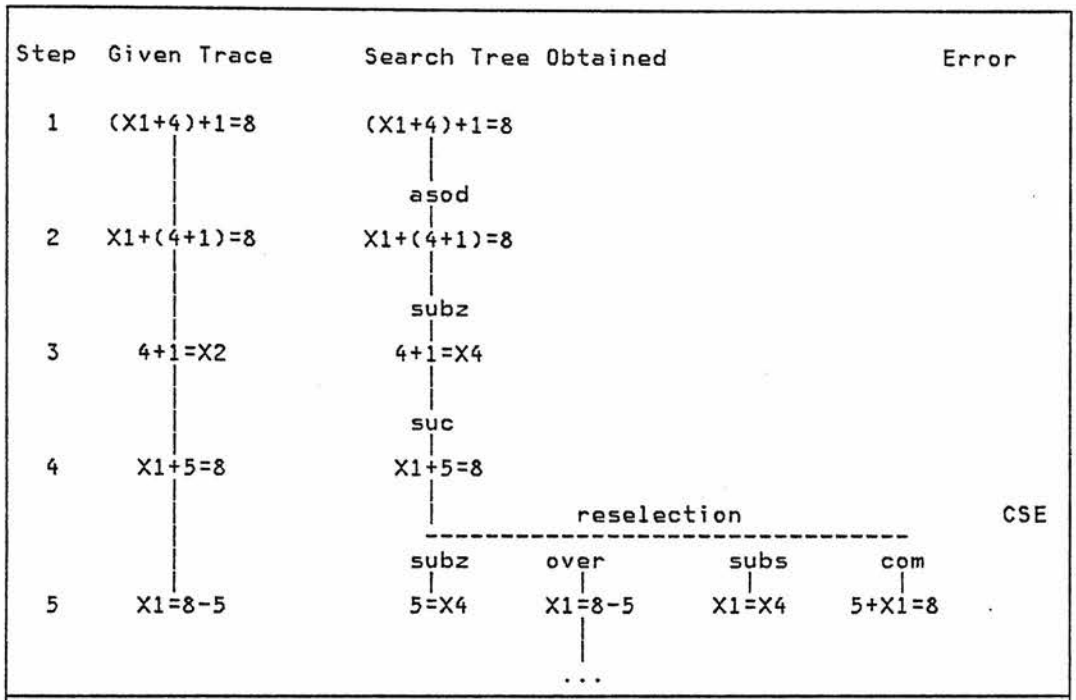


Fig. 1.12 Search for the Solution of (X1+4)+1=8

The conflicting selection error detected in step 4 is corrected by conflict resolution. The following clause, however, is <u>not</u> generated.

```
over2:  X1+X2=X3        <-   X2=:5     & !     & X1=X3-X2.
```

With this clause the error would recur again, if a similar equation was given to the system (eg. equation (X1+6)+1=8). The system detects that the error has, in fact, recurred and takes a special action to avoid further problems. The method used is discussed in chapter 3.10. The following clause is generated by the system:

```
over2:  X1+X2=X3   <-   var(X1)   & int(X2)   & !   & X1=X3-X2.
```

With this clause the equation '(X1+6)+1=8', for example, can be solved without errors. More details about all this can be found in chapter 3.14. We also show there how various other problems are dealt with by the system. After all of these problems have been solved the system is capable of solving many other similar equations.

In our experiments we have limited our attention to equations containing one variable only. Moreover, all terms must contain '+' as the function symbol. Its subterms must be similar: They may contain integers, or similar terms as subterms. However, the system can be taught to solve different types of equations as well.

1.4.3 Learning Division in the Extended System

Division is much more difficult to learn than, say, addition or subtraction. The algorithm which we have investigated cannot be learned without the use of GS clauses referring to various goals awaiting solution. This is why our system ELM1 which does not use such clauses cannot learn to do division.

The examples presented here are described in chapter 4.6 in detail. Our experiments with ELM2 started with addition. We verified that the system could learn addition without difficulties. Division was done next.

The clauses used in these experiments are shown in the following figure.

```
Clauses Used for Addition:
--------------------------
subs1: X1+X2=X3        & Gs   <-   X1=:(X5+X6)  & ! & X1=X4 & X4+X2=X3
asoc:  X1+(X2+X3)=X4   & Gs   <-   !  & (X1+X2)+X3=X4
subs:  X1+X2=X3        & Gs   <-   !  & X1=X4    & X4+X2=X3
subz:  X1+X2=X3        & Gs   <-   !  & X2=X4    & X1+X4=X3
eq:    X1=X1           & Gs   <-   !
pred:  ...
suc :  ...

Clauses Added:
--------------
izero:  X1=X2          & Gs   <-   !  & 0+X1=X2
subsd:  X1/X2=X3       & Gs   <-   !  & X1=X4   & X4/X2=X3
distd: (X1+X2)/X3=X4   & Gs   <-   !  & (X1/X3 + X2/X3)=X4
cancd:  X1/X1=X2       & Gs   <-   !  & 1=X2
```

Fig. 1.13 Clauses Used for Division

```
suc > subs1 > asoc  >  subz > subs
                       subz > eq
                       pred > eq
                       subz > izero
                       pred > izero
```

Fig. 1.14 Priority Orderings Used

Each clause used contains a conjunction in the clause head. Symbol Gs represents a predicate or a conjunction of predicates. Each clause can be selected only if the clause head matches the current goals.

Clause 'subsd', for example, can be selected with the goals '4/2=X1 & write(X1)' which were given to the system to solve. Predicate 'X1/X2=X3' from the head of clause 'subsd' matches the first of the two goals, namely '4/2=X1'. Variable 'Gs' matches the second goal, that is 'write(X1)'. The new goals obtained as a result of applying this clause are '4=X2 & X2/2=X1'. The following figure shows how the search continues.

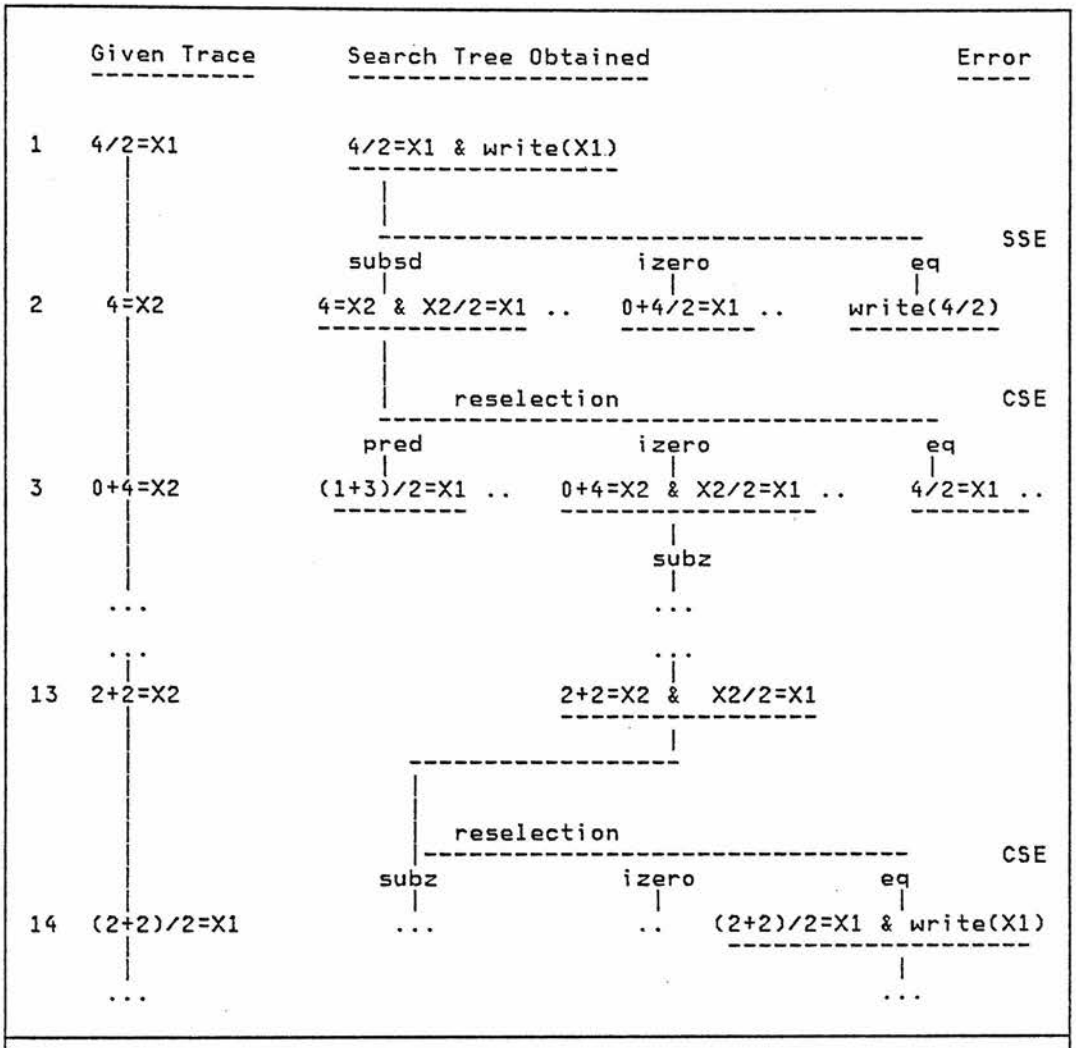


Fig. 1.15 Search for the Solution of 4/2=X1 (Phase 1)

Fig. 5.6 in chapter 5.6 shows how the search continues after step 14. The last goal is 'write(2)' indicating that '2' is the result.

Error Correction

All conflicting selection errors are corrected by the method of conflict resolution. The basis of the method is described in chapter 3. Chapter 5 explains how conflict resolution is used in the extended system ELM2.

Four new clauses were generated as a result of correcting the errors detected in this example. The new clauses are shown in the following figure.

```
izero1: X1=X2    & Gs <-  ( X1=:4 v
                           Gs=:((../..=..) & ..)  v
                           Gs=:((X2/..=..) & ..))
                           & !  & 0+X1=X2

eq1:    X1=X1    & Gs <-  (X1=:(2+..)  v  X1=:(X2+X2)  v
                           X1=:(X3+..) &  Gs=:((../X3=..) & ..)  v
                           X1=:(..+X4) &  Gs=:((../X4=..) & ..)) & !

subs2: X1+X2=X3 & Gs  <-  X1=:(../..) v  X2=:(../..)  & ! &
                           X1=X4       &  X4+X2 =X3

eq2:   X1=X1     & Gs  <-  (X1=:1 v
                            Gs=:((X1+.. = ..)   & ..) v
                            Gs=:((..+(../..)=..) & ..) ) & !
```

Fig. 1.16 Clauses Generated (*)

```
  suc > subs1 >  asoc > subz                  > subs
                        subz                  > eq
                                        pred  > eq

          eq1 > subs2 > subz > izero1 > pred  > izero
                                        subsd > izero
                                        subsd > eq
                               eq2    >  pred
```

Fig. 1.17 Priority Orderings Generated

After all errors have been dealt with, the problem 4/2=X1 was given to the system again to check that all errors have been eliminated. We found that all old errors were eliminated, but three new errors were introduced: two simple selection errors and one conflicting selection error.

The first simple selection error occurred because the system failed to add 'subz > izero1' when clause 'izero1' was generated. The second

(*) Symbol '..' represents a unique variable. It may be replaced by any variable 'Xi' not occurring elsewhere in the clause.

simple selection error was similarto the first one.

The conflicting selection error occurred in step 18. The following figure shows what happened:

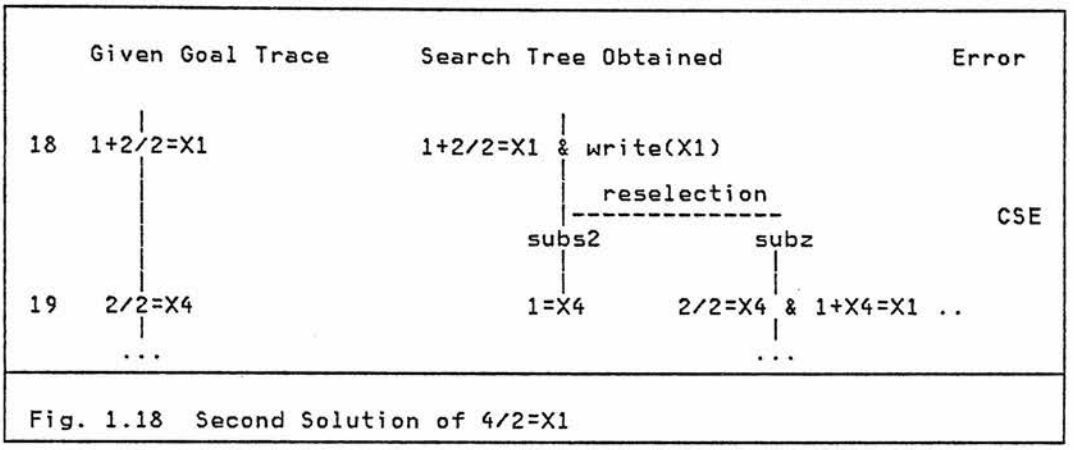


Fig. 1.18 Second Solution of 4/2=X1

We notice that no error was detected in this step before, when the problem 4/2=X1 was solved the first time. To correct the error the system generated clause 'subz1':

```
subz1:  X1+X2 =X3  & Gs  <-   int(X1)  &  !  &  X2=X4  & X1+X4=X3
```

Clause 'subz1' was given priority over clause 'subs1'.

We were interested to see whether any errors would occur if a different problem of division was solved, and we chose the problem '9/3=X1'. We found that no errors occurred during the solution of this problem. Two of the existing clauses were modified using the method of 'learning from examples'. The modified clauses are shown in the following figure.

```
izero1: X1=X2     & Gs  <-   Gs=:((../..)=.. & ..) & !  & 0+X1=X2
eq1:    X1=X1     & Gs  <-   X1=:X3+.. & Gs=:((../X3=..) & ..) & !
```

After all these modifications have been performed the system is capable of solving both problems of division shown before without errors.

1.4.4 Dealing with Various Letter Series

Our program can predict the next letter in a given letter series. The task of letter series prediction was studied by others. Simon and Kotovsky have written their program in 1961. However, their program is a special purpose program and could not learn to solve simple equations like ours. Waterman's program (1970) is more general even though it requires the use of a 'special purpose heuristic' which allows the program to make intelligent guesses about the size of the group of letters that are repeated. The examples shown here are described in chapter 4.7 in more detail and in chapter 6 we explain how our work is related to Waterman's.

In our system each particular series is represented by a term. Series 'aabb', for example, is represented by '((a:a):b):b'. The symbol '-' used later represents a blank. Only one clause was available to the system initially:

```
s0: series(X1) <- ! & write(X2)
```

The meaning of this clause is this: If series 'X1' is given the next letter in the series is 'X2'.

Letter Series 'aabb'.

The following figure shows several examples of a particular letter series. Each series shown was given to the system as a 'goal'. Error detection was performed, and if any errors were detected error correction followed. Errors occurred with the first two goals only.

```
Problem:    Given Goal :

 1          series (-:a)                    )  Errors occurred
 2          series ((-:a):a)                )  with these goals

 3          series (((-:a):a):b)            )  The next letter was
 4          series ((((-:a):a):b):b)        )  correctly obtained
```

Fig. 1.19 Goals Given to the System

The following figure shows the first error detected. We see that goal obtained by the system differs from the corresponding goal in the given goal trace. The variable 'X2' should have been instantiated to 'a', and because of that an 'instantiation error' is detected.

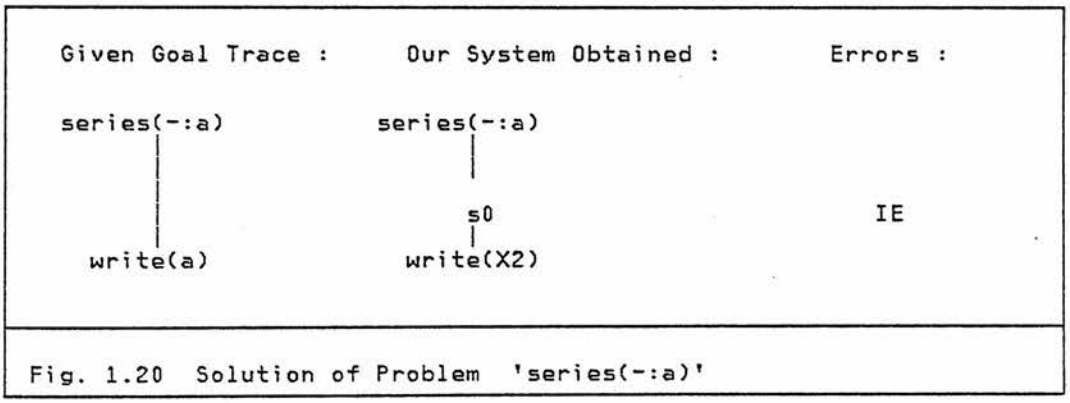


Fig. 1.20 Solution of Problem 'series(-:a)'

The instantiation error is corrected by generation of clause 's1', which is a new version of clause 's0'. The new clause was given priority over clause 's0'. The new clause was as follows:

```
s1: series(X1)  <- (a=:X2  v X1=:(..:X2))  & ! & write(X2)
```

Clause 's1' contains new 'variable instantiating predicates'. In this series of experiments new variable instantiating predicates were chosen from the following set:

- Xi=:Xj described before,
- next(Xi,Xj), true if 'Xj' is the next letter after 'Xi'.

Chapter 4 explains in detail how 'variable instantiating predicates' are introduced by the system.

The following figure shows what happened after the second example of this letter series was given to the system.

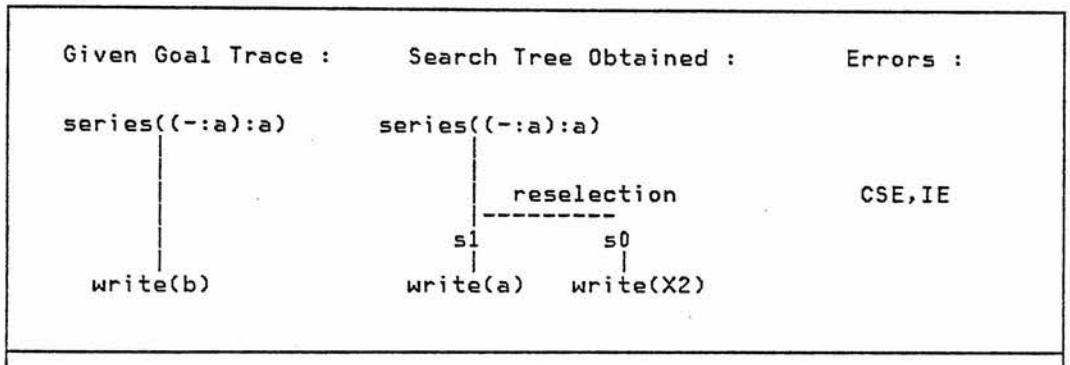


Fig. 1.21 Search for a Solution

First, clause 's1' was selected and so a selection error was detected. Reselection was performed and then instantiation error was detected. The instantiation error was corrected by generation of clause 's2' which is a new version of clause 's0'. The selection error was corrected by further modifications of the same clause. The following clause was obtained as a result:

```
s2: series(X1) <-  X1=: ((..:L2):..)  & next(L2,X2) & ! & write(X2)
```

Clause 's2' uses the second letter from the right (L2) to instantiate variable 'X2' which represents the next letter in the series (*). This variable is instantiated to the next letter after 'L2'. After the new clause has been added to the existing clauses, the remaining two goals shown in Fig. 1.19 were solved without errors. Variable 'X2' was correctly instantiated in each case. The system has learned to predict the next letter in this series.

Letter Series 'abmcdm'

This letter series was dealt with in a similar way. The same clause was given to the system initially:

```
z0: series(X1) <- ! & write(X2).
```

The same type of predicates were used to modify this clause and the new clause versions generated. The goals given to the system here were as follows:

No.:	Given Goal :	
1	series (-:a)	)
2	series ((-:a):b)	)
3	series (((-:a):b):m)	) Errors occurred
4	series ((((-:a):b):m):c)	) with these goals
5	series (((((...):b):m):c):d)	)
6	series (((((...):m):c):d):m)	)
7	series (((((...):c):d):m):e)	) The next letter
8	series (((((...):d):m):e):f)	) was correctly
9	series (((((...):m):e):f):m)	) predicted

Fig. 1.22 Goals Given to the System

Errors occurred with goals 1-6. After all of these errors have been corrected, the correct letter was predicted afterwards.

(*) Each variable Li used here was chosen to represent a particular letter in the given series; L1 represents the last letter, L2 represents the letter just before L1 etc.

Some of the clauses which were generated by the system are shown in the following figure. (Three other 'intermediate' versions which were also generated by the system appear in chapter 4.6). Clauses 'z5' and 'z6' were given a higher priority than clause 'z4' (z5>z4 and z6>z4).

```
z4:  series(X1) <-   (X1=:(((..:L3):..):L1) & next(L3,L1)) v ...) &
                      X1=:          (..:L1) & next(L1,X2)
                      & !                   & write (X2)

z5:  series(X1) <-  ( X1=:((..:L2):L1)      & next(L2,L1) &
                      m =:X2  & !           & write (X2)

z6:  series(X1) <-  ( X1=:(..:m)  v
                      X1=:(((..:L3:L2):..)  & next(L3,L2) ) &
                      X1=:    ((..:L2):..)  & next(L2,X2)
                      & !                   & write (X2)
```

Fig. 1.23 Some Clauses Obtained by the System

All three clauses shown are capable of extending various examples of the series 'abmcdm' correctly.

Clause 'z4', for example, uses the last letter in the given series to derive the value of the next letter. Clause 'z4' can extend the series 'cdme' (goal no.7) correctly. The next letter will be 'f'.

Clause 'z5' instantiates the next letter in the series to 'm'. This clause will extend the series 'dmef' (goal no.8) correctly.

Clause 'z6' uses the second letter before the end to derive the value of the next letter. This clause will extend the series 'mefm' (goal no.9) correctly. The next letter will be 'g'.

Various constraints and priority orderings which the system has generated ensure that the right clause is selected with each example of the series 'abmcdm'.

1.5 IMPLEMENTATION

Our system was implemented in PROLOG. We used the DEC-10 machine in Edinburgh and the DEC-10 machine in Dundee.

The problem solving subsystem and the error detection subsystem which are described in chapter 2 were fully implemented.

The error correction subsystem was also implemented. The method of correction of simple selection errors and conflicting selection errors was implemented as described in chapter 3.

The system implemented can also correct instantiation errors. The method is described in chapter 4. The clause instances required in the process have to be supplied manually. A relatively simple extension is needed so that the clause instances are supplied automatically by the system.

The extended system ELM2 described in chapter 5 was partly implemented. The system can generate new constrained clauses correctly, but the selection and rejection contexts required have to be manually supplied. The experimental results presented in chapter 5.6 were obtained by simulating the system on paper.

2 SEARCH AND DETECTION OF ERRORS

2.1 INTRODUCTION

In this section we discuss how our system searches for a solution of the given problem. We describe how clauses are selected for each given goal and how they are applied. Also, we explain how goal traces are used in detection of errors. The detection of several different types of error is discussed.

Detection of errors is accompanied by storage of the relevant information. This information is used for error correction which is described in detail later. It is the main topic of the rest of the thesis.

Clause Selection

For each given goal a number of clauses can be selected from among the existing clauses. Various conditions affect selection. Each particular clause can be selected only if it matches the current goal. The 'predicate constraints' must also be satisfied, and there must not be any other clause with 'higher priority' whose selection conditions are also satisfied. The selection conditions are discussed in detail in section 2.

Clause Application

All clauses selected are applied, and so, search proceeds in parallel on different branches. Application of each clause selected creates a new node in the search tree. Each new node generated contains a list of subgoals yet to be solved.

Detection of Errors

The goal traces given enable our system to detect errors. The goals in the goal trace are compared with the corresponding goals obtained by the system, and error detection is performed on the basis of the comparison. If the goals mismatch, a selection error is detected. Depending on whether the goals on at least one of the branches are correct, the error is classified either as a 'simple selection error', or a 'conflicting selection error'. If the goals obtained are not instantiated as they should be, an instantiation error is detected. Detection of these different types of errors is described in detail in section 4.

Reselection and Backtracking

Certain errors indicate that the solution cannot be reached by following the search any further from the current node. If such an error is detected, reselection is performed. During reselection the clauses which were selected before are ignored. Reselection may be repeated. If all possible clauses have been tried out without success, backtracking is initiated. During backtracking the system goes back to the previous node in the search tree and performs reselection there. More details about reselection and backtracking can be found in section 5.

2.2 CLAUSE SELECTION

The objective of clause selection is to consider various clauses in a given set and choose some, according to various conditions. The clauses selected are then applied.

All clauses are initially assumed to be candidates for selection. Initially all of these clauses appear in the set S, a set of clauses to be applied. Some of the clauses may be deleted from this set by the selection algorithm, as the following figure shows.

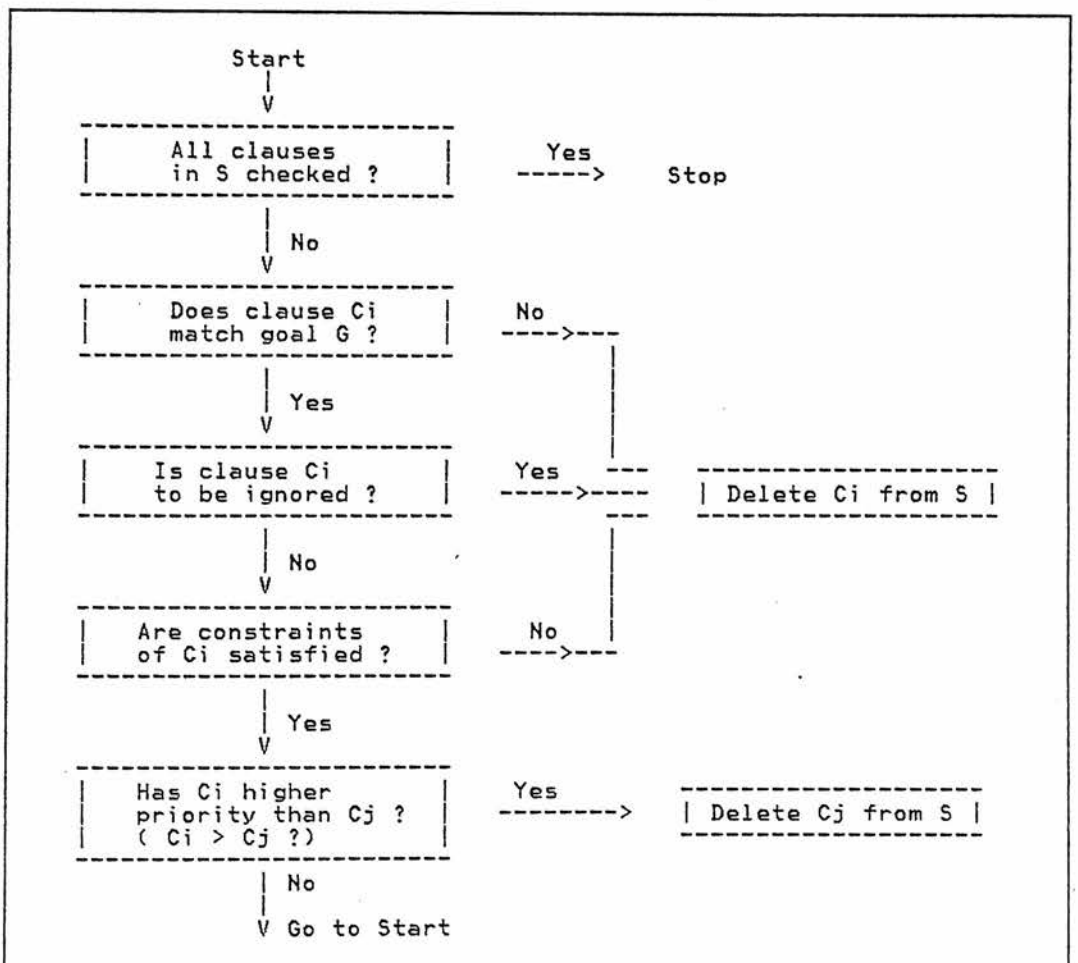


Fig. 2.1 Selection Algorithm

In the following we shall describe how various conditions affect deletion of clauses from the set S.

Matching

Each clause can be selected only if it matches the current goal. If the clause considered does not match the current goal, the clause is deleted from the set S. The predicate constraints of this clause are ignored.

If the clause considered matches the given goal, the system checks if the predicate constraints are also satisfied.

In our system clauses are written all in the same form P <- Q. We say that clause P <- Q matches some goal G if P matches G. Predicate P is sometimes referred to here as the 'clause head'.

Why are some Clauses Ignored

Certain clauses may be ignored during selection so as to enable the system to perform reselection. The clauses to be ignored are simply deleted from S.

In section 5 dealing with reselection we shall describe how the system determines which clauses are to be ignored.

Predicate Constraints

Predicate constraints are predicates whose truth-value is tested during selection. If the predicate constraints are not satisfied the clause considered is deleted from the set S.

The clauses in our system may contain a number of such constraints, appearing as disjuncts or conjuncts in a more complex expression. Selection of each clause depends on the truth-value of this expression. The constraints are easily identified since they appear before (to the left of) the symbol '!'.

Example

The predicates Q1,Q2 and Q3 in the clause C shown below represent predicate constraints, since they appear before the symbol '!'. Clause C can be selected only if the expression '(Q1 v Q2) & Q3' is 'true'.

```
P <-  (Q1 v Q2)  & Q3   & !   & Q4
```

The predicate constraints used in our examples are special in that their truth or falsity is always easily established. No search is required in their solution. In general search might, however, be required in this phase.

The search in the selection phase is related to a 'look-ahead' whose purpose is to assess whether the clause considered is likely to lead to a successful solution of the given goal(s).

Priority Orderings

Priority orderings affect selection of clauses satisfying conditions previously mentioned. Use of priority orderings may result in further deletions of clauses from S. Let us consider this in more detail.

Suppose that a number of clauses satisfy the selection conditions previously mentioned. Priority orderings specify which of those clauses should be retained in S, and which ones should be deleted. The clauses with 'higher priority' are retained and the clauses with 'lower priority' are deleted.

Priority ordering Ci > Cj, for example, specifies that if all selection conditions of clause Ci are satisfied, clause Cj should be deleted from S.

We notice that deletion of lower priority clauses may be performed irrespective of whether they actually match the given goal or whether their constraints are satisfied.

The ordering of clauses in our system is transitive. If, for example, all the other selection conditions of clause Ci are satisfied and Ci > Cj and Cj > Ck exist, both clause Cj and clause Ck are deleted from S. It is not important whether clause Cj actually matches the given goal, or whether its constraints are satisfied.

Discussion

In some systems clauses are ordered implicitly, as in PROLOG, for example (Warren, 1977). Clauses of PROLOG are written in the order top down, and this is also the order in which selection is attempted. The production rules used by Waterman (1970b) are also ordered implicitly this way.

The use of explicit priority orderings has certain advantages. Firstly, it is possible to simulate both breadth-first and depth-first search in the system. If no orderings exist, then the system performs a breadth-first search. If priority orderings are added so that the clauses would be totally ordered, the system performs a depth-first

search.

The use of priority orderings facilitates correction of certain errors. New priority orderings can be added as need arises. Also, the danger that two or more clauses would be repeatedly reordered can be avoided. The introduction of new priority orderings will be discussed in chapter 3.

2.3 CLAUSE APPLICATION

After all clauses have been either selected or rejected, the system performs clause application. All clauses which have been selected are applied. Clause application involves the following steps.

First, the head of each clause is matched against the current goal. Afterwards the predicate constraints are 'solved' as if they were goals. In our experiments these predicates were solved without any search. That is no subgoals were generated as a result of solving them.

After the predicate constraints have been solved, the predicates in the clause body are introduced as new 'goals' to be solved. This introduces a new node in the search tree.

All clauses selected are applied like this. Care is taken so that expansion of one branch would not affect other branches as well. In our implementation this is achieved using the following technique. As each new node is being created, the variables in it are systematically replaced by new variables not occurring elsewhere. Then instantiation of variables in one node do not affect variables in other nodes in any way.

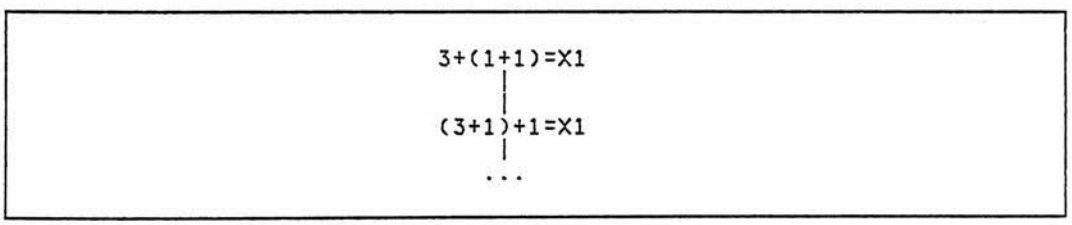
We chose this technique since it is very easy to implement. Unfortunately, however, some errors cannot be detected as a result. In chapter 4.5 we explain why this happens and explain how the problems could be overcome.

2.4 DETECTION OF SELECTION ERRORS

Use of Goal Traces

After each clause selected has been applied the system tries to detect errors. This is done on the basis of given 'goal traces'. The goal traces show how the given problems are to be solved. It is a sequence of goals which would have been produced by a perfect, error free system.

If, for example, goal '3+(1+1)=X1' were supposed to be transformed into '(3+1)+1=X1' on the basis of associativity, the goal trace would contain the following goals:



In our system each goal trace contains only one goal irrespective of whether a conjunction of goals is being solved. That is if a conjunction of goals is being solved the first conjunct only is shown in the trace.

In our system the goal traces are generated automatically, using clauses. This saves us the work of typing each

individual goal in. The clauses used for this purpose cannot be retrieved or used in any way by other parts of the system.

Error Detection

Errors are detected on the basis of comparison of goals obtained by the system with corresponding goals in the given goal trace. Errors are detected if the goals differ. The following types of errors may be detected:

- simple selection error (SSE),
- conflicting selection error (CSE),
- instantiation error (IE).

Detection of selection errors will be discussed in the following in more detail. Detection of instantiation errors will be discussed in one of the following sections.

Simple Selection Errors

Let us consider the conditions for detection of simple selection errors in more detail. Simple selection errors arise

- if two or more clauses (eg. Ci,Cj) have been selected for a particular goal;
- if the goals Gi obtained by application of clause Ci 'differ' from the goals Gt;
- if the goals Gj obtained by application of clause Cj do not 'differ' from goals Gt.

Here, goals Gt represent the appropriate goals in the given goal trace.

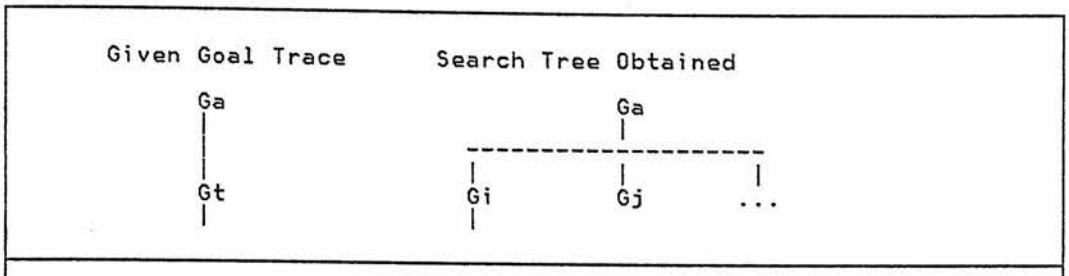


Fig. 2.2 Detection of Simple Selection Errors

The goals Gi are regarded as 'different' from 'Gt' if:

- goals Gi do not match goals Gt or
- goals Gi are more specific than goals Gt.

Goals Gi are more specific than the goals Gt if they can be obtained from Gt by, say, substituting some of the variables in it by constants. If goals Gi are more general than goals Gt then an instantiation error (IE) is detected instead.

Detection of simple selection error affects further search. Further expansion on the branch in error is terminated. It is assumed that this branch would not lead to the solution.

Example

Detection of simple selection errors is illustrated in Fig. 2.3.

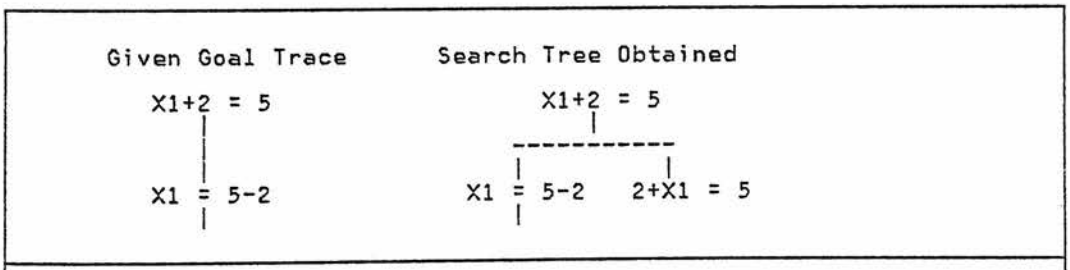


Fig. 2.3 Example of a Simple Selection Error

The current goal to be solved in our example is 'X1+2=5'. The search tree

obtained by the system shows two branches indicating that two clauses have been selected. The goals obtained by application of the clauses selected are shown at the tips of the two branches shown.

As one of the goals obtained 'differs' from the corresponding goal in the given goal trace while the other does not, a 'simple selection error' is detected. No attempt is made to try to solve the subgoal '2+X1=5' which 'differs' from the corresponding goal in the goal trace. Further search on this branch is terminated.

-*-

After the search for a solution has terminated error correction is performed. Correction of simple selection errors is performed by rearranging the existing priority orderings for the clauses involved. The details of this will be discussed later (in chapter 3.3).

Conflicting Selection Errors

Conflicting selection errors are detected in the following situation:

- if one or more clauses have been selected for some goal and
- if the goals obtained by application of all the clauses selected 'differ' from the corresponding goals in the given goal trace.

We have already mentioned that goals Gi and Gt 'differ' if they do not match, or if Gi is more specific than Gt.

Conflicting selection errors are more difficult to deal with than the simple selection errors. Firstly, corrective action has to be taken during selection to enable the system to find the solution. As all the goals obtained by the system differ from the corresponding goals in the goal trace, we

assume that it has no sense to continue in the search any further. The system has to select clauses which have not been selected before. We call this process reselection. Backtracking may also be initiated.

Example

Detection of conflicting selection errors is illustrated in Fig. 2.4.

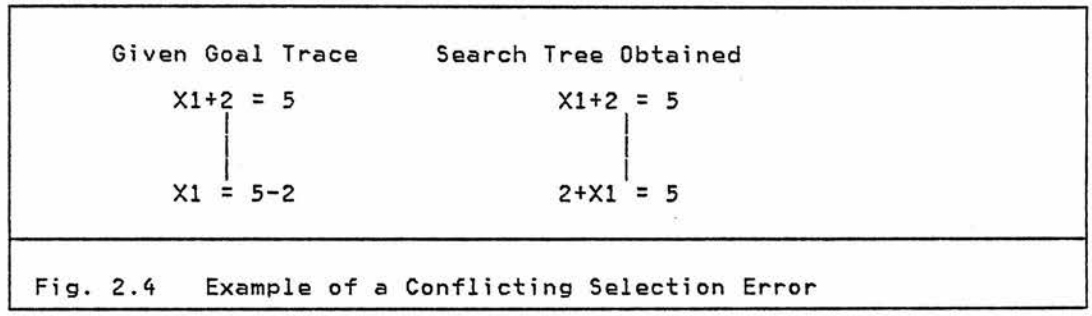


Fig. 2.4 Example of a Conflicting Selection Error

The goal to be solved in our example is 'X1+2=5'. There is only one clause selected for this goal. As the goal obtained by application of the clause selected mismatches the corresponding goal in the goal trace given, a 'conflicting selection error' is detected. No attempt is made to solve the subgoal '2+X1=5'. The system performs reselection with the goal 'X1+2=5'.

-*-

Conflicting selection errors are corrected by the method of 'conflict resolution'. Each particular conflicting selection error is corrected by generation of one new clause. The priority orderings may also be rearranged. The method is referred to as a 'conflict resolution'. Most of chapter 3 is devoted to this topic.

2.5 RESELECTION AND BACKTRACKING

Reselection is performed whenever a conflicting selection error has been detected. The purpose of reselection is to select some other clauses which have not been selected before so as to find the solution of the given problem.

Whenever a conflicting selection error is detected with some particular goal, the clauses currently selected are added to the set of clauses to be ignored which is kept by the system. When selection is performed the next time the clauses to be ignored are deleted from the set of clauses dealt with (set S). Clauses are selected from among the remaining clauses only. The priority orderings among these clauses are respected, and so only the clauses with the highest priority are selected.

The set of clauses to be ignored during selection is associated with a particular goal in a particular node. Selection of clauses in other nodes is not affected by this.

If all clauses available to the system have been tried out but the error has not yet been eliminated, backtracking is initiated. During backtracking the system goes back to the preceding node in the search tree and performs reselection for that node. The search is then continued in the forward direction. If after this another error is detected, the system backtracks further. Backtracking and search forward is repeated until all conflicting selection errors have been eliminated, or until there are no possibilities left.

If all alternatives have been tried out, and the conflicting errors have not been avoided, search is terminated. The answer 'NO' is given, indicating that the given problem cannot be solved.

2.6 DETECTION OF INSTANTIATION ERRORS

Instantiation errors are detected if the goals obtained by the system are not fully instantiated. The goals in the given goal trace serve as a norm.

Example

Detection of instantiation errors is illustrated in the following figure.

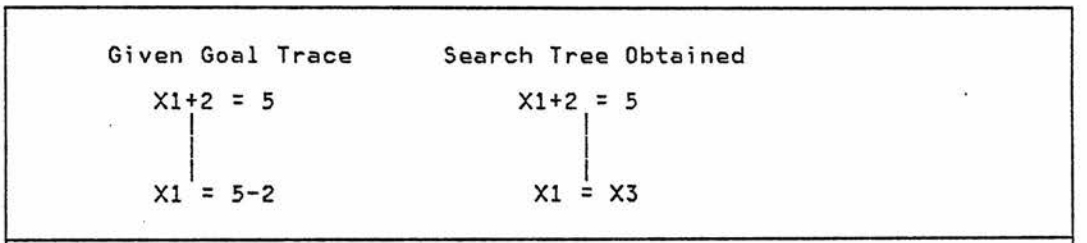


Fig. 2.5 Example of an Instantiation Error

The goal to be solved in this example is 'X1+2=5'. There is only one clause selected for this goal by the system. As the goal 'X1=X3' is an uninstantiated version of the corresponding goal in the given goal trace, an 'instantiation error' is detected.

We notice that a conflicting selection error cannot be detected here, since the goal 'X1=X3' does not 'differ' from the corresponding goal in the goal trace given. (The explanation of what happens when two goals differ is given on page 51.)

After the error has been detected, the current goal is instantiated in the same way as the corresponding goal in the goal trace given. This is achieved by matching the current goal against the corresponding goal in the goal trace given. Thus the current goal becomes 'X1=5-2'.

Detection of instantiation errors is not merely a matter of comparison of the corresponding subterms in the two goals being compared. The error in the Fig. 2.6 could not be detected this way. The goal obtained by the system contains two distinct variables (X1,X2), but the corresponding goal in the goal trace given contains two occurrences of one variable (X1).

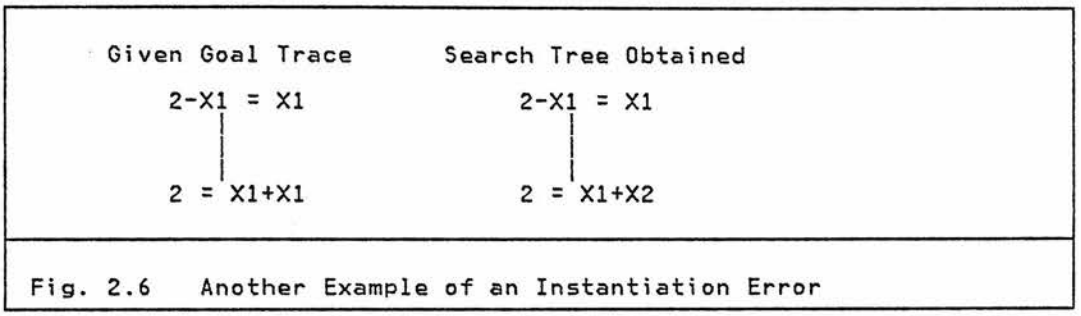


Fig. 2.6 Another Example of an Instantiation Error

In order to detect instantiation errors of this type the following method of detection is used in our system. Let Gs represent the goals obtained by the system, and Gt represent the corresponding goals in the goal trace given. The system performs the following:

- it checks if Gs and Gt match,
- it replaces all distinct variables in Gs by constants
- it checks again if Gs and Gt match.

If any variables in Gs have been replaced by constants the match of Gs and Gt will be prevented. So, if the goals do not match, an instantiation error is detected. This way the system can detect if two or more variables are used in some goal instead of several occurrences of one variable, as in Fig. 2.6.

Instantiation errors are corrected by addition of variable instantiating predicates to the existing clauses. The details of how this is done will be discussed in chapter 4.

2.7 SUMMARY

In this chapter we have described how the system searches for a solution. We have discussed clause selection and application. We have explained how the search tree is expanded.

We have mentioned that a number of conditions affect selection of clauses. For example, they must match the given goal, the predicate constraints must be satisfied, and there must not be any clause with higher priority that satisfies all these conditions. All clauses selected are applied.

We have explained how errors are detected. We have mentioned that the goals in the given trace are compared with the corresponding goals obtained by the system. Errors are detected on the basis of this comparison. Selection errors are detected if the goals mismatch. Instantiation errors are detected if the goals match, but if they are not instantiated as they should be.

The goals belonging to one node in the search tree are never compared with other goals belonging to other nodes. Consequently certain types of instantiation errors cannot be detected by the system. In chapter 4.5 we explain how the system can be extended so that it could detect all types of instantiation errors.

Some Extensions

In our system the goal traces contain many goals, irrespective of whether they are needed or not. Thus detection of errors is relatively easy. It would be interesting to consider how errors could be detected in more difficult conditions - for example, if the goal trace given contained fewer goals than it does at the moment.

Information could also be provided 'on demand'. That is the system should be able to ask for more information if it got into trouble while solving problems. More work is need in this area, however, to establish when exactly should the system demand more information, or how the questions should be formulated, for example.

3 CORRECTION OF SELECTION ERRORS

3.1 INTRODUCTION

In this chapter we shall describe a phase of error correction which is performed after the search for the solution of the given problem has terminated. No errors are ever corrected while the search still continues. This is because the system has not yet had a chance to establish which clauses should be selected in each step.

In this chapter we shall explain how selection errors are corrected by the system. Correction of both 'simple selection errors' and 'conflicting selection errors' will be discussed in detail. The detection of these errors was discussed in chapter 2.4.

Objectives of Error Correction

In section 2 we explain what the main objectives of error correction are. The main objective is, of course, to eliminate the current error. However, the modifications performed should not affect solutions of old problems. That is, old errors should not be reintroduced by the modifications performed. Also, the solutions of new problems should be facilitated as much as possible.

Correction Simple Selection Errors

Simple selection errors are corrected by addition of new priority orderings for the clauses involved. The method is described in detail in section 3.

Correction of one error may affect other errors as well. Some errors may disappear altogether, but others may be more difficult to correct. Because of this we have to check what type of error we have, if any, prior to attempting to correct it. If the simple selection error changed into a conflicting selection error appropriate action is taken, as described in the following.

Correction of Conflicting Selection Errors

Conflicting selection errors are corrected by the method of conflict resolution. The method is described in section 4. More details are given later.

The main objective of conflict resolution is to prevent the introduction of conflicting priority orderings. To avoid the introduction of such priority orderings the system generates one new clause which is a constrained version of one of the existing clauses. The new clause is given a higher priority than the clause(s) selected by mistake.

New clauses are generated on the basis of selection and rejection contexts. These are the contexts in which the clause dealt with should be selected, or rejected from selection. A description of how these contexts are obtained is given in section 4.

The aim of the system is to constrain the clause so that the clause cannot be selected in the 'rejection context'. The

'selection context' is also taken into account. It should always be possible to select the new clause in this context.

Each clause may be constrained either by modification of the existing constraints, or by introduction of new constraints. The system always tries to modify the existing constraints first. Let us first see how new constraints are introduced.

Generation of Constraints

New constraints are obtained in the following way. The given set of predicates are scanned and tests are made which ones, if any, can be used as constraints. Various subterms obtained by decomposition of selection and rejection contexts are used in the process. A particular predicate is used as a new constraint if it is 'true' when used with the selection context subterms and 'false' when used with the rejection context subterms.

The constraints which are more general are preferred to constraints which are more specific. This minimizes the chances of generating overconstrained clauses.

If several constraints are found they are all used in a disjunctive expression generated. Disjunctions of constraints are preferred since they are more general than each individual constraint.

The constraints introduced by our system as it is implemented, may contain at most two variables. This could be extended.

Modification of the Existing Constraints

Clauses can also be constrained by modifications of the existing constraints. If the clause to be constrained contains one or more disjunctions of constraints, an attempt is made to modify them. New constraints are added only if the clause cannot be constrained this way. The method is discussed in section 7.

Both selection and rejection contexts are used by the system in the process. Selection contexts help to 'clean up' the existing constraints generated before. Rejection contexts are used afterwards. The aim is to modify the existing disjunctions of constraints so that selection of the clause dealt with would be prevented in these contexts.

Dealing with Multiple Conflicts

The number of rejection contexts considered during conflict resolution varies. It depends on how many clauses there are which have priority higher than Cj, the clause which should have been selected, and priority lower than Ci, the clause selected by mistake. These 'multiple conflicts' that is conflicts which involve a number of clauses are resolved by generating one new clause only. The new clause is a constrained version of clause Cj. All rejection contexts identified are taken into account. Details are discussed in section 8.

Some other possible ways of dealing with multiple conflicts are also discussed in this section.

Reorganization of Priority Orderings

After the conflict resolution has been performed, additional priority orderings are sometimes introduced. All priority orderings which include both the old clause and the new version generated are examined. The aim is to make them as similar as possible and this is why new priority orderings are sometimes introduced. The introduction of such priority orderings eliminates possible errors. The method is explained in section 9.

Preventing Recurrence of Errors

It is unlikely that all clauses will always be correctly constrained since only a limited amount of information is used in the process (ie. particular selection and rejection contexts). This, we believe, has to be accepted as unavoidable. It is important, however, that clauses are not again constrained in the same way. To avoid this we do the following.

Whenever some clause version Ci'' is being generated we check if another version Ci' has been generated before. If certain conditions are satisfied a particular context is identified as 'additional selection context', and used in the process of generating Ci''. The additional selection contexts help to generate constraints which are general.

The method mentioned is described in detail in section 10. An example is given in that section showing how this method helps to prevent an error from recurring.

Learning from Examples

Modifications of clauses need not only be instigated by errors. In our system modifications are sometimes performed after the new clauses generated have been used in new contexts. This helps to simplify the constraints generated before. Also, future errors may be prevented.

The method of 'learning from examples' is described in section 11. We point out that the existing disjunction of constraints may be modified by deletion of some of the disjuncts. The selection contexts encountered are used to identify the disjuncts to be deleted.

Experimental Results

In section 14 the experimental results are given. We show how various problems have been solved by our system. Then we show which errors were detected, and describe how these errors were corrected.

The problems examined deal with addition and subtraction of integers and simple equation solving. Fourteen problems have been analyzed and many of the results are presented here.

More experimental results are presented in chapters 4.7 and 5.6. Many of the errors detected there are also corrected by the method of conflict resolution.

3.2 OBJECTIVES OF ERROR CORRECTION

Our main objective is to consider the selection errors detected and make modifications to the existing clauses so that these errors would not recur.

However, the modifications should be done so that solutions to old problems would not be affected. That is, new errors should not be introduced when old problems are solved.

Also, the modifications should be done so that various new problems will be easily solved. That is, the modifications introduced should facilitate solutions of problems which are similar, but not quite the same as the problems already solved.

In summary our three objectives are:

- to eliminate the error dealt with,
- that solutions of old problems should not be affected,
- that solutions of future problems should be facilitated.

Different systems will exhibit different learning progress depending on whether these objectives are satisfied and to what extent. This is why the objectives are important.

When designing our system we tried to ensure that the first objective was always satisfied. That is whenever a new clause was introduced by the system, we checked that the current error was eliminated. This was always achieved with the problems listed in section 14, for example.

The second objective mentioned was also the second one on our priority list. Whenever some modifications have been performed by the system we checked that the solutions of the old problems would not be affected by this. If they were we

analyzed the situation carefully and then tried to amend our program.

Due to basic assumptions adopted, we could not ensure that the old problems would always be solved without errors. This is because after each problem has been solved, only a limited amount of information is stored for future reference. Examples of this are rare, however. None of the problems listed in section 14 suffered from this fate. Problem 4/2=X1, for example, whose solution is described in chapter 5.6, had to be given to the system twice in succession before all errors were eliminated.

As far as the third requirement is concerned, we found it difficult to decide if it was ever actually satisfied. However, we found it useful to keep this objective in mind all the time, and in particular when we were comparing two different versions of our system.

If with one version of the system we got less errors than with another one, we were interested in finding out why this was so. If correction of one error in this version has eliminated other errors as well, then we had a reason for believing that this version was 'better' than the other one. Obviously a more thorough analysis has to be performed before any conclusions are made.

3.3 CORRECTION OF SIMPLE SELECTION ERRORS

The Method in Principle

Simple selection errors arise if a number of clauses have been selected in some context and if only one of those clauses

should have been selected there. Their detection was discussed in chapter 2.4.

Simple selection errors are corrected by introduction of new priority orderings so that only the correct clause will be selected in future.

Let us suppose that an error has occurred because two clauses were selected, instead of just one. Suppose, for example, that clauses Ci and Cj were selected instead of just clause Ci. To correct this a priority ordering Ci > Cj is introduced, specifying that priority should be given to clause Ci over clause Cj during selection. Selection of clause Cj is conditional. An attempt to select this clause is made only if clause Ci cannot be selected.

If several clauses have been selected instead of just one the error is corrected in a similar way. If clauses Ci,Cj .. Cn were selected instead of just clause Ci, n-1 priority orderings would be introduced. Each ordering gives priority to clause Ci over one of the other clauses involved.

Introduction of Priority Orderings

In our system no errors are corrected as long as search continues. This is because the system has not yet had a chance to establish which clauses should be selected in each step. However, the relevant information about the errors detected is kept by the system so that these errors could be corrected later. With selection errors the error information stored includes the following:

- current context (some goal G),
- clause selected (Ci).

Examples of such error information are given later (see

section 14 with the experimental results).

The error information is 'passed' from one node to another as the search tree is being expanded. The current node contains information about all errors on the current branch of the search tree. The error information associated with the solution path is used in the error correction phase. If the error information contains clause Ci and context (goal) G, clause Ci represents the 'correct clause' which should be selected with this context.

The error information stored enables the system to introduce the appropriate priority orderings. Selection is performed with the context G stored, and if more than one clause has been selected, new priority orderings are introduced. Priority is given to clause Ci over any other clause selected.

Each new priority ordering Ci > Cj is stored together with the context G as follows:

G: Ci > Cj

The contexts stored are used in correction of conflicting selection errors. We shall see that in the next section. They are not used during clause selection in any way.

Side-effects of Introduction of Priority Orderings

Correction of errors may have side-effects. Correction of one error may eliminate the need for correcting other errors. If, for example, two errors are to be corrected by introduction of priority ordering Ci > Cj, the priority ordering does not have to be introduced twice. Similarly, the

priority orderings which can be derived from others by transitivity do not need to be introduced, too.

Unfortunately, however, correction of one error may also introduce further difficulties. The next selection error may be more difficult to correct. Under certain circumstances a simple selection error may change into a conflicting selection error.

Example

Suppose we are dealing with two errors which were classified as simple selection errors at the time of their detection. Suppose that the first error was corrected by the introduction of Ci > Cj. This priority ordering prevents selection of clause Cj. If this is just the clause that should be selected next, a conflicting selection error arises. So, a simple selection error can change into a conflicting selection error.

	Problem Solution Phase:			Error Correction Phase:		
Step No.	Desired Clause	Clauses Selected	Type of Error	Clauses Selected	Type of Error	Priority Ord. Added
1	Ci	Ci,Cj	SSE	Ci,Cj	SSE	Ci>Cj
2	Cj	Ci,Cj	SSE	Ci	CSE	

Fig. 3.1 Side-effects of Error Correction

The problems of side-effects are dealt with in the following way. Before any selection error is corrected a check is made to see whether this error has already been eliminated, or whether the type of the error has changed. If the simple selection error has changed into a conflicting selection error, an appropriate action is taken. The details will be given in the next section.

3.4 CORRECTION OF CONFLICTING SELECTION ERRORS

In this section we shall describe how conflicting selection errors are corrected by the system. We shall not concern ourselves with their detection here. Detection of these errors was discussed in chapter 2.4.

In this section we shall give an overview of our method and more details will be given in the following few sections.

Conflicting Priority Orderings

Conflicting selection errors cannot be corrected in the same way as simple selection errors. If they were corrected that way a system of conflicting priority orderings would be generated. We would find that some clause Ci would have both higher and lower priority than some clause Cj. The results of selection would be unpredictable.

Conflicting System of Priority Orderings:	Ci > Cj Cj > Ci

The system of conflicting priority orderings which could be obtained can involve more than two clauses:

Another Conflicting System:	Ci > Ck > Cj Cj > Ci

Conflict Resolution

In the following we shall describe how conflicting

selection errors are corrected without the introduction of conflicting priority orderings. The method used will be referred to as conflict resolution. In this section we shall show how we deal with conflicts which involve two priority orderings only. The more general case will be discussed later (in section 8).

Let us suppose that a conflicting selection error has occurred because after priority ordering Ci > Cj has been introduced, clause Cj was supposed to be selected. We have already mentioned that correct selection of clause Cj cannot be achieved by introduction of the priority ordering Cj > Ci. What we do is introduce a priority ordering Cj'> Ci. Clause Cj' is a new version of clause Cj whose selection has been suitably constrained. Later in this section we shall explain how this clause is generated.

The old clause Cj is kept by our system. This enables the system to generate other constrained versions of this clause later if this was necessary. The following figure shows how the priority orderings are modified by the system.

Before conflict resolution:	After conflict resolution:
Ci > Cj	Cj' > Ci > Cj
(Cj > Ci)	---

Fig. 3.2 Priority Orderings

Role of Rejection and Selection Contexts

The constraints of clause Cj' are generated so that selection of clause Ci would not be affected by the introduction of the ordering Cj'> Ci. The new constraints of

Cj' should prevent the selection of this clause in the selection context of clause Ci. This context is referred to here as the rejection context. This is because clause Cj' should be 'rejected' from selection in that context.

The selection context of clause Cj' must also be taken into account when the new constraints are generated. That is it should be possible to select clause Cj' in its selection context.

Let us now see how the selection and rejection contexts are obtained by the system.

The rejection context is the context associated with the priority ordering Ci > Cj. The system has no problem with obtaining this context. We have mentioned earlier that whenever some priority ordering is introduced, the associated context is stored with it (see page 68).

The selection context is the context which is associated with the particular error dealt with. The 'error information' stored contains the context needed. The following information is associated with each error:

- the name of clause Cj
- the associated context G.

The context G is used as the selection context here.

Generation of New Constraints

The new clause Cj' is generated on the basis of an analysis of particular selection and rejection contexts. Let us call these contexts S and R.

We have said that the aim of the modifications is to prevent

selection of clause Cj' in R, but at the same time not prevent selection of this clause in S. The requirements may be achieved either by modification of the existing cdnstraints or by introduction of new constraints.

If clause Cj already contains some constraints, the system tries to modify them. The method will be described in section 7. If clause Cj does not contain any constraints, or if the existing constraints cannot be modified, new constraints are introduced. In the following we shall explain how this is done.

New constraints are chosen from a given set of predicates. The same set is normally used with the whole series of related problems. To generate the new constraints of clause Cj various subterms of this clause are considered together with the corresponding subterms of S and R. Then various predicates from the given set are tried out in turn.

The predicate chosen is used as a constraint if it is true when used with the subterm(s) of S, and if it is false when used with the corresponding subterms of R.

Each constraint chosen in this way will prevent selection of Cj' in R, because it will be 'false' in that context. However, it will still be possible to select this clause in the selection context S.

The process of constraint generation does not stop after one constraint has been found. Our system tries to find all possible predicates satisfying the conditions above. However, to limit the number of the constraints generated more general constraints are preferred to the constraints which are more specific (see section 6). All constraints found are used as disjuncts in the final expression generated.

Use of the Constraints Generated

Let us see how the new constraints are used to modify the clause dealt with.

The new constraints are always added to the existing predicates in the clause as conjuncts. If the clause contains some constraints already the new constraints are added in front of (to the left of) the existing constraints. Thus the predicates added last are considered first during clause selection.

If the clause does not contain any constraints the new constraints are added to the left of the symbol '!'. This symbol is used in our system to identify the constraints. The predicates to the left of the symbol '!' are treated as constraints.

Example

In this example we shall describe how one conflicting selection error is corrected by the system. Full details about how this error arose can be found in section 14, where some of our experimental results are given. The error occurred in step 4 of the solution of the problem of addition 3+2=X1'.

The error detected was corrected by the modification of the following clause:

```
subs: X1+X2=X3  <-  ! & X1=X4 & X4+X2=X3
```

The following selection and rejection contexts were used in the process:

Selection context S: (3+1)+1=X3

Rejection context R: 3+2=X3

The details about how these contexts were identified can be found in section 14. Both contexts shown were analyzed by the system together with the head of clause 'subs'. Various subterms obtained in the process are shown in the following table:

Subterms of Clause Head:	Subterms of Sel.Context S:	Subterms of Rej.Context R:
X1+X2	(3+1)+1	3+2
X1	3+1	3
X2	1	2
X3	X3	X3

Fig. 3.3 Various Subterms Considered

The following predicates were tried out as constraints:

int(Xk), var(Xm) and Xi=:Xj.

The explanation of what these predicates mean can be found on page 108. Let us see how new constraints were generated on the basis of the subterms shown in Fig. 3.3.

One of the subterm of S used was '3+1'. The corresponding subterm of R is '3'. The program looked for a predicate P(X1) such that P(3+1) would be 'true' and P(3) 'false'. Predicate int(X1), for example, does not satisfy these requirements. Predicate 'X1=:X4+X5', on the other hand, satisfies both requirements. That is '(3+1)=:(X4+X5)' is 'true', and '1=:(X4+X5)' is 'false'. The predicate 'X1=:X4+X5' is one of the constraints found by the program.

The search for new constraints continued like this. The constraint 'X1=3+X5' was not generated, however, since this constraint is more specific than the constraint 'X1=:X4+X5' generated before. In section 6 we shall explain how the generation of such constraints is prevented.

One more constraint was found by the system later: X2=:1. This

constraint was added as a disjunct to the constraint found before. The following disjunction was obtained as a result:

$$X1=:X4+X5 \quad v \quad X2=:1.$$

This disjunction was used to modify clause 'subs'. It was added as a conjunct left of the symbol '!'. The new clause generated was as follows:

```
subs1: X1+X2=X3  <-  (X1=:(X5+X6)  v X2=:1)  & ! & X1=X4 & X4+X2=X3
```

-*-

Introduction of Relational Constraints

Relational constraints, that is constraints containing two variables, are obtained by relating two sets of subterms. The new set of subterms is related to another set of subterms obtained earlier. The following subterms are always stored together:

Ti - the clause head,
Xi - the variable currently dealt with,
Si - corresponding subterm of S,
Ri - corresponding subterm of R.

Predicate P(Xi,Xj) is introduced as a constraint, if P(Si,Sj) is 'true', and P(Ri,Rj) is 'false'.

Various predicate constraints which may be found are added as disjuncts to other constraints generated earlier.

Example

In this example we shall show how relational constraints are generated. The selection and rejection contexts will be given. The following predicates will be considered as constraints here:

next(Xi,Xj) and Xi=:Xj.

Predicate next(Xi,Xj) is true, if 'Xj' is the next letter after 'Xi'. The meaning of the predicate 'Xi=:Xj' is explained on page 108.

Suppose that the clause head of the clause to be constrained is as shown in the following. Suppose that the following selection and rejection contexts are also available:

Clause Head:	series ((X1:X2) : X3)
Selection Context S:	series ((a:a) : b)
Rejection Context R:	series (((a:a):b) :c)

Let us now see how the constraints are generated. Various sets subterms of contexts S and R are considered by the system in turn. An attempt is made to generate new constraints at various points.

Predicate 'X1=:X2', for example, is chosen as a constraint for the following reasons. Predicate 'a=:a' constructed from the subterms of S is 'true' and predicate '(a:a)=:b' constructed from the corresponding subterms of R is 'false'. Predicates 'X2=:X1' and 'next(X2,X3)' are used as constraints for similar reasons.

The experimental results presented in chapters 4.7 and 5.6 show how various relational constraints were generated by the system in the process of correcting errors.

More details about various aspects of the process of constraint generation will be given in the following sections.

3.5 DECOMPOSITION OF CONTEXTS

In the previous section we have explained how the selection context S and the rejection context R are used in the process of generation of new constraints. In this section we shall give more details about how various subterms of S and R are obtained by the system.

The subterms of contexts S and R are obtained by a process of decomposition. Decomposition may be repeated several times. Three terms are always dealt with at the same time. One is from the clause head and the others are the corresponding subterms in S and R (*).

Decomposition is quite straightforward if all three terms contain the same function symbol and the same number of arguments (**). Decomposition produces a list of arguments for each term. The following example will illustrate this.

Subterms of Clause Head:	Subterms of context S:	Subterms of context R:	
X1+X2	(3+1)+1	3+2	) Original terms
X1	(3+1)	3	) Subterms
X2	1	2	) obtained

Fig. 3.5 Example of Decomposition of Terms

The subterms obtained by decomposition may be decomposed further. A number of factors determine whether the three subterms currently considered should be decomposed into other subterms, or whether this process should be terminated. The decision depends on what all three subterms are. Normally decomposition continues until the clause head subterm

(*) Clauses and their constituents are treated as 'terms' here.

(**) If f(X1,..Xn) is the term dealt with 'f' is the function symbol.

obtained is either a variable, or a constant.

If the clause head subterm is a variable the system tries to generate new constraints in the way described in the previous section. Further decomposition may follow.

If the clause head subterm is a constant further decomposition of this term and the corresponding subterms of S and R is terminated.

Decomposition of Variables

If the clause head subterm dealt with is a variable the system tries to generate new constraints. However, further decomposition of the terms dealt with may follow. The subterms of S and R dealt with determine what happens.

If both subterms dealt with are terms with the same function symbol (eg. +) decomposition is continued. Let us see what happens to the variable obtained by decomposition of the clause head.

The variable is replaced by a term, obtained by generalization of the corresponding subterm of S. The main function symbol is preserved, but new variables are introduced in all the argument places.

The relationship between the original variable (X) and the term replacing it (T) needs to be established. In our system this is done using a predicate 'X=:T' which we call a decomposition predicate. When this predicate is 'solved' the system checks if the terms 'T' and 'X' are identical. If they are not the system tries to instantiate the term 'T' to make them identical.

The decomposition predicate is added to any constraint P(T) which might be found later. That is the conjunction 'X=:T & P(T)' is used in constraining the clause dealt with.

Example

Let us see how the following terms are decomposed into subterms. Suppose the subterm of the clause head is a variable, and the corresponding selection and rejection context subterms are as shown in the following figure.

Subterms of clause head:	Subterms of context S:	Subterms of context R:	
X0	(3+1)+1	3+2	) Original terms
X1	3+1	3	) Subterms
X2	1	2	) obtained

Fig. 3.6 Another Example of Decomposition of Terms

First, variable 'X0' is replaced by a term 'X1+X2', obtained by generalization of '(3+1)+1'. The main function symbol, that is '+' in this case, is preserved and new variables are introduced in the two argument places. This term is decomposed into subterms X1 and X2.

The relationship between the original clause head subterm 'X0' and the subterm 'X1+X2' is established using the following 'decomposition predicate': X0=:X1+X2'.

-*-

3.6 HOW GENERAL CONSTRAINTS ARE GENERATED

In section 4 we have described how individual constraints are generated. Let us see now how the system attempts to generate the 'most general constraints'.

We believe that it is desirable that the system should generate constraints which are as general as possible. More general constraints seem better than the specific ones, since the chances of generating overconstrained clauses are minimized. Also, the introduction of more general constraints has fewer side-effects than the introduction of more specific ones. This is because the introduction of more general constraints represents the least possible change that can be made which eliminates the error considered.

Let us see what the system does to generate these 'general constraints'.

Firstly, disjunctions of constraints are generated from the individual constraints found. The disjunctions of constraints are more general than each individual constraint.

Secondly, each individual constraint is generated so that it would be as general as possible.

Example

In one of our examples presented earlier we have shown how constraints were generated on the basis of the following information (see the page 74):

Clause Head : X1+X2 = X3

Selection Context S: (3+1)+1 = X3

Rejection Context R: 3+2 = X3

The following constraint was generated as a result:

X1=:X4+X5 v X2=:1

This constraint is more general than the following constraints:

X1=:3+X5 v X2=:1	X1=:X4+X5
X1=:X4+1 v X2=:1	X1=:3+X5
X1=:3+1 v X2=:1	X1=:X4+1
	X2=:1

All of the constraints shown satisfy the conditions mentioned before: They are 'false' in the context R and 'true' in the context S.

-*-

How the General Constraints are Found

In order to minimize the number of constraints in each disjunction the system tries to exclude those constraints which are more specific than others. Two rather particular methods are used by the system.

Method No 1

The predicates to be tried out as constraints are considered in a certain order. Predicates which are more general than others are considered before the predicates which are more specific. If one of these more general predicates is chosen as a constraint, the predicate(s) which are more specific are eliminated from the set of constraints to be considered.

Example

If the predicate int(X1) is chosen as a constraint the predicate of X1=:1 (or X1=:2 etc.) is eliminated from the set of constraints to be considered, because predicate 'int(X1)' is more general than 'X1=:1'. We

see that 'int(X1)' is 'true' whenever 'X1=:1' is 'true'.

-*-

Method No 2

Once some constraints are found for a particular set of subterms, further decomposition of the subterms involved is 'restricted'. For the moment let us assume that the process of decomposition is actually terminated. A more precise explanation will be given later.

First, let us see an example showing how this method prevents generation of specific constraints.

Example

Suppose that constraints are to be generated on the basis of the following information:

Clause Head :		X1+X2 = X3
Selection Context	S:	(3+1)+1 = X3
Rejection Context	R:	3+2 = X3

Once the constraint 'X1=:X4+X5' is found, further decomposition of the subterms 'X4' and 'X5' is 'restricted', together with the decomposition of the corresponding subterms in S and R. Consequently, the constraint

X1=: X4+X5 & X5=:3

is not generated. This constraint is more specific than the one already generated.

-*-

The method just described is quite effective in preventing the generation of constraints which are more specific than the ones already generated. The method prevents generation of various constraints of type (2) below after the constraint (1)

has been generated:

(1) X=: T(Xi) T(Xi) ... some term

(2) X=: T(Xi) & P(Xi) P(Xi) ... some predicate

The method described needs to be improved, however, since it can also prevent the generation of constraints which are not more specific than the ones already generated. For example, if the constraint (3) shown in the following was found by the system, further decomposition of the subterms involved would be terminated. The constraint (4) shown in the following would not be generated. In general this constraint is <u>not</u> more specific than the constraint (3).

(3) X=: T(Xi) & P(Xi)

(4) X=: T(Xi) & Xi=: T'(Xj) & P'(Xj)

<u>Restricted Decomposition</u>

After the decomposition of a particular set of terms has been 'restricted', a new phase of decomposition is initiated. This phase of decomposition of terms enables our system to generate various relational constraints. It enables the system to generate the constraint (4), for example, after the constraint (5) has been generated.

(5) X=: T(Xi) & P(Xi)

(6) X=: T(Xi) & Xi=: T'(Xj) & R(Xj,Xk)

The 'second order subterms' which are obtained during the secondary phase decomposition are used in conjunction with other subterms, obtained by the ordinary process of decomposition.

3.7 MODIFICATION OF EXISTING CONSTRAINTS

Clauses need not necessarily be constrained as described before, that is by addition of new constraints. Sometimes it is possible to modify the existing constraints and achieve similar effects. The advantage of the latter approach is that the resulting clauses are simpler than they would have been if new constraints were added.

In the following we shall describe how the existing constraints are sometimes modified by the system.

The aim of all the modifications is to constrain selection of the clause dealt with so that it could not be selected in any of the rejection contexts identified. Any modifications performed should not, however, affect selection of this clause in its selection context. The same requirements were already mentioned in section 4 showing how new constraints are generated.

So, the existing constraints are modified on the basis of selection and rejection contexts. The selection context is used by the system first. Later we shall explain why.

Use of Selection Contexts

The selection context is used in the following way. The system attempts to modify all the existing disjunctions by deleting those disjuncts which are false in this context. This helps to 'clean up' the constraints generated before. The system is, in effect, 'learning from examples' at this stage (see section 11).

Use of Rejection Contexts

Each rejection context is used in a similar way. The system tries to delete those disjuncts which are true in the rejection context considered. The process is somewhat more complex and therefore we shall describe it a more detail in the following.

The existing disjunctions are scanned through, one by one. The system tries to find one which can be modified so that its truth-value would change from 'true' to 'false' in the rejection context considered. It tries to delete those disjuncts which are 'true' so that the resulting expression would be 'false'. If by chance all disjuncts were to be deleted the resulting expression would be 'empty' and therefore not 'false'. Such modifications are not allowed.

If no disjunction of constraints can be modified the system attempts to generate new constraints. The method was already described before (in section 4).

Example

In this example we shall describe one phase in the process of generation of clause 'subz1' which was generated by our system. More details about how this clause was generated can be found in section 14, where some of our experimental results are given.

Let us start with the following clause version here:

```
X1+X2=X3  <- (var(X1)  v X2=:(3+..)  v int(X3)) & !
                & X2=X4  & X1+X4=X3
```

Let us see how the constraints of this clause are modified. Let us use

the following contexts in the process:

Selection context: X1+(3+1)=7

Rejection context: (X1+3)+1=7

We see that none of the constraints is 'false' in the selection context and so the disjunction does not need to be modified at this stage. However, one disjunct is 'true' in the rejection context (predicate int(7)). This is why the predicate 'int(X3)' is deleted from the existing disjunction of constraints. The following clause is obtained as a result:

```
subz1: X1+X2=X3  <- (var(X1)  v X2=:(3+..)) & !
                     & X2=X4  & X1+X4=X3
```

This clause is simpler than the original clause. If the original clause was modified by addition of new constraints the following clause would have been obtained:

```
X1+X2=X3  <- (var(X1) v X2=:(3+..) v int(X3)) &
             (var(X1) v X2=:(..+..)) &
              & !     & X2=X4     & X1+X4=X3
```

We see that this clause is more complex than the one obtained by the system.

-*-

Why is the Selection Context Used First

We have mentioned before that any modifications performed should not affect selection of the clause dealt with in its selection context. This requirement is also satisfied after the existing constraints have been modified. The system

cannot obtain a clause which could not be selected in its selection context. The truth-value of a modified disjunction can never be 'false' if it was not 'false' before. This is because all the disjuncts which could have been 'false' in the selection context were deleted first.

If the rejection context(s) were used first and some disjuncts were deleted as a result, the remaining disjuncts could be 'false' in the selection context. The clause dealt with would not satisfy the requirement mentioned before. The clause could not be selected in its selection context.

We see that it is rather important that the selection context is used as described before the rejection context.

3.8 DEALING WITH MULTIPLE CONFLICTS

In this section we shall continue in the explanation of how conflict resolution is performed. We shall describe how the system deals with conflicts which involve any number of priority orderings. The resolution of conflicts which involve two priority orderings was discussed in section 4.

The method of conflict resolution is the same in principle, irrespective of how many priority orderings there are. In each case we can identify the clause Cj which should have been selected, but was not. A new version of this clause is generated by the system and given priority higher than each clause Ci selected instead (using Cj'> Ci).

Clause Cj' is constrained so that selection of other clauses will not be affected by the introduction of new priority orderings. In other words it is constrained so that it cannot

be selected instead of any other clause involved in the conflict.

Suppose, for example, that the conflict to be resolved involves clauses Ci1,..Cin. Let us assume first that these clauses are not ordered by priority orderings.

Before conflict resolution:		Afterwards:
Ci1 > Cj		Cj'> Ci1 > Cj
...		...
Cin > Cj		Cj'> Cin > Cj
Cj > Ci1	) Conflicting	
...	) orderings	
Cj > Cin	) are not added	

Fig. 3.7 Conflict with Several Priority Orderings

Clause Cj' is constrained so that it will not be selected instead of any one of the clauses Ci1..Cin. That is clause Cj' is constrained so that it will not be selected in any one of the contexts associated with the orderings Ci1 > Cj .. Cin > Cj.

If the clauses Ci1..Cin are ordered clause Cj' is generated in a similar way. Suppose, for example, that the clauses are ordered as shown in the following figure. Clause Cj' is constrained so that it will not be selected instead of any one of the clauses Ci1..Cin. It is not sufficient to consider the selection of clause Ci1 only, because clause Cj' could easily be selected in the context of clause Cin, for example. In this context we expect the clause Cin to be selected.

Before conflict resolution:		Afterwards:
Ci1 > .. > Cin > Cj		Cj'> Ci1 > .. > Cin > Cj
Cj > Ci1	<-- Conflicting ordering is not added	

Fig. 3.8 Conflict with Several Priority Orderings

We see that clause Cj' is generated in the same way irrespective of whether the clauses Ci1..Cin are ordered among themselves. If the clauses Ci1..Cin are only partially ordered clause Cj' is still generated in the same way. Clause Cj' is constrained so that it will not be selected instead of any one of the clauses involved. The following figure shows an example of a set of clauses which is partially ordered.

A Partially Ordered Set of Clauses:	Ci1 > Ci2 > Ci3 > Ci5 > Cj Ci2 > Ci4 > Ci5

To resolve each multiple conflict the set of 'conflicting priority orderings' is identified first. These orderings show how the clauses Ci1..Cn and Cj are ordered among themselves. The contexts associated with these priority orderings are then used as 'rejection contexts' by the system.

Each rejection context Ri is used together with the selection context S in the way described in section 4. We have mentioned before that the system tries to modify the existing constraints first. If selection of the clause dealt with cannot be constrained this way, new constraints are introduced.

New constraints are not introduced in a blind manner. The system always tries to establish whether new modifications are really necessary when it is dealing with a particular rejection context. The system tries to select clause Cj' in

the particular rejection context dealt with and if the clause Cj' cannot be selected, no modifications are made at this stage. The clause Cj' may already be sufficiently constrained as a result of using other rejection contexts before.

The rejection contexts are processed one by one in the way described. After all these contexts have been considered the clause Cj' is added to the existing clauses.

Other Ways of Dealing with Multiple Conflicts

Let us now see some other possible ways of resolving conflicts which are different from the method used in our system. This section may be skipped by the reader who wishes to find out how our system works. As we shall see the conflicts can be resolved by reorganization of the existing priority orderings and by constraining various clauses accordingly.

Suppose a conflict arose because the priority ordering Cj > Ci was to be introduced. To avoid the conflict the system would add Cj'> Ci instead, and then generate clause Cj', a constrained version of clause Cj. The priority ordering Cj > Ci can, however, be added as it is, provided one of the other conflicting priority orderings is deleted and used in its place. Deletion of Cx > Cy, for example, will lead to generation of Cx', a constrained version of Cx.

We notice that it is also possible to constrain selection of several clauses whose priorities are 'similar'. That is instead of choosing one clause, giving it priority and constraining it, a group of clauses can be chosen and used in a similar way. First, the conflicting set of priority orderings needs to be split up into two groups A and B. The following figure shows how this could be done.

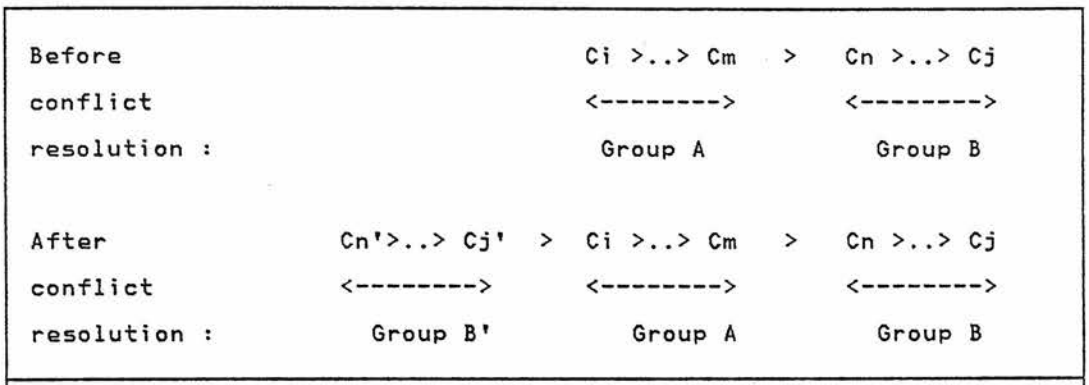


Fig. 3.9 Conflict with Several Priority Orderings

Next, group B' is generated and given priority over group A. That is the clauses which have the lowest priority in B' are chosen and given priority over the particular clauses from group A - the clauses with the highest priority in that group. Also, each clause in the group B' is constrained so that it will not affect selection of clauses which appear to have lower priority now.

So far we have not found any reason why we should adopt one of these more complex methods of conflict resolution. The method which we have described before is simpler and this is why it is used in our system.

3.9 REORGANIZATION OF PRIORITY ORDERINGS

Let us now see how the priority orderings are sometimes rearranged by the system so that the new clause generated by conflict resolution would be correctly included among the existing clauses. If this was not done selection errors could occur with some of the old problems.

We have mentioned before that each new clause generated by conflict resolution is given priority over the other clauses involved in the conflict (see section 4). Apart from this the priorities among clauses should not really change.

That is if a certain priority relationship between clauses Cx and Cy existed before the conflict was resolved then a similar relationship should also exist afterwards. This is why sometimes additional priority orderings are generated by the system. They reestablish a particular relationship which existed before.

Suppose that the conflict dealt with was resolved by generating clause Cj' in the way described in section 4. No action needs to be taken to ensure that the clause Cj' remains the high priority clause with respect to other clauses.

For example, let us suppose that before conflict resolution he relationship Cj >> Cx existed (*). We see that the relationship Cj'>> Cx will exist also after the conflict has been resolved and clause Cj' generated. That is Cj'>> Cx is true, because Cj'>> Cj and Cj >> Cx are true.

However, if Cx >> Cj existed before the conflict was resolved the relationship Cx >> Cj' will not necessarily exist afterwards. So in this case new priority orderings are sometimes introduced by the system giving clause Cj' appropriate priority. Let us consider an example.

(*) The relation '>>' is transitive. That is Cx >> Cz is true if Cx > Cz is true, or if Cx > Cy and Cy >> Cz are true.

Example

Let us assume that the conflict to be resolved involves the priority orderings shown in the following figure.

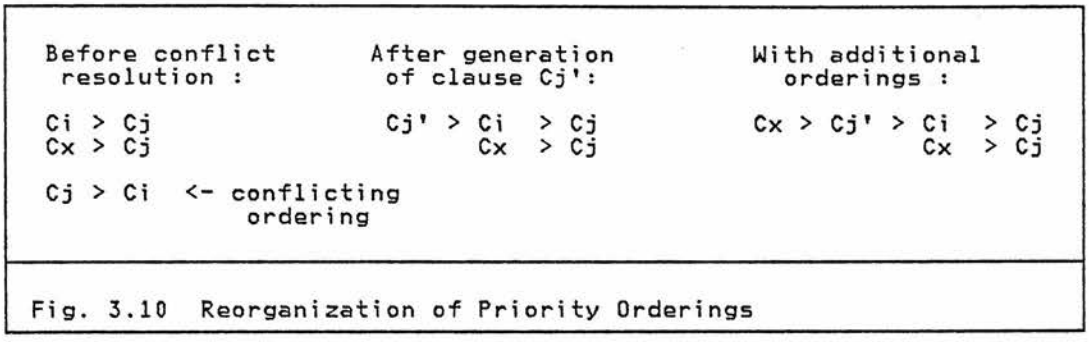


Fig. 3.10 Reorganization of Priority Orderings

The ordering Cx > Cj' is the additional ordering generated by the system. It is introduced because before the conflict was resolved the ordering Cx > Cj existed too.

-*-

We have mentioned before that the aim of generating the additional priority orderings is to prevent selection errors - if some of the old problems were to be solved again. If no errors could be expected to occur in the old contexts then the orderings do not really need to be added. This is what the system tries to establish.

Suppose the system is considering the addition of the ordering Cx > Cj'. To check whether any errors could occur in the old contexts it performs a selection in the context associated with the ordering Cx > Cj. If it finds that clause Cx only is selected no new orderings are added. However, if clause Cj' is selected as well the priority ordering Cx > Cj' is added. This prevents a potential simple selection error in that context.

For practical reasons the system does not consider the addition of all possible orderings Cx > Cj' that we could

find. It considers the addition of the ordering Cx > Cj' if it can find an ordering Cx > Cc, where Cc is one of the clauses involved in the conflict.

In our previous example the ordering Cx > Cj' was added, because Cx > Ci existed and clause Ci was one of the clauses involved in the conflict (Ci=Cc). Also, we have assumed that an error would have occurred if selection was performed in the context associated with the ordering Cx > Ci.

In one of the earlier versions of our system we tried to prevent all possible errors by addition of all possible orderings of the type Cx > Cj'. However, this did not seem particularly useful. No errors were introduced with the problems listed in section 14 when we made the system to consider only some of the ordering Cx > Cj', as described. Also, the system seemed to be taking a relatively long time to find all the relations of the type Cx > Cj' and perform selection in the associated contexts.

3.10 PREVENTING RECURRENCE OF ERRORS

It is unlikely that new clauses will always be correctly constrained. This is because when errors are being corrected, only limited information is used in the process (ie. particular selection and rejection contexts). This, we believe, has to be accepted. However, similar errors should not be repeated too many times.

If some clause has been incorrectly constrained this clause should not be constrained again in the same way, when a similar error is encountered later. The error should not recur repeatedly.

Example

The following figure shows a part of the search tree obtained by the system when it was solving the equation (X1+3)+1=7. The example is dealt with in detail on page 122.

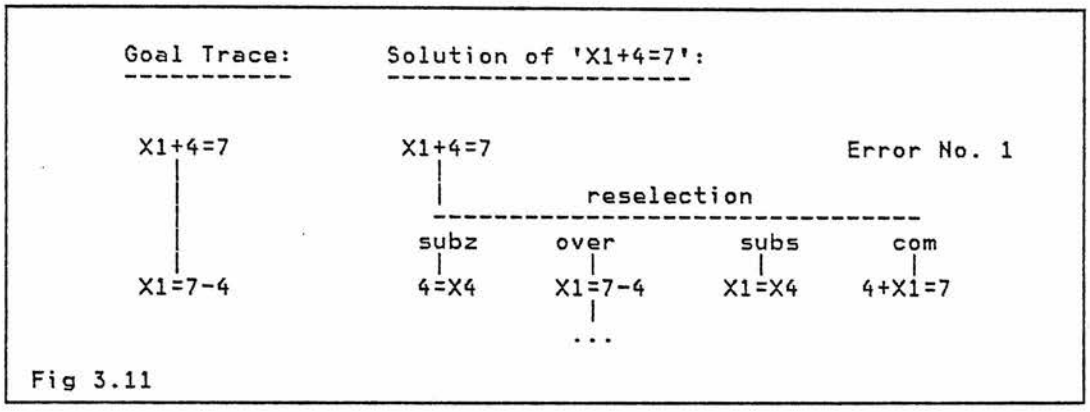


Fig 3.11

The error shown above was corrected by the method of conflict resolution. The new clause generated was, however, incorrectly constrained. This is why a similar error occurred later, when the subgoal X1+5=8 was dealt with by the system.

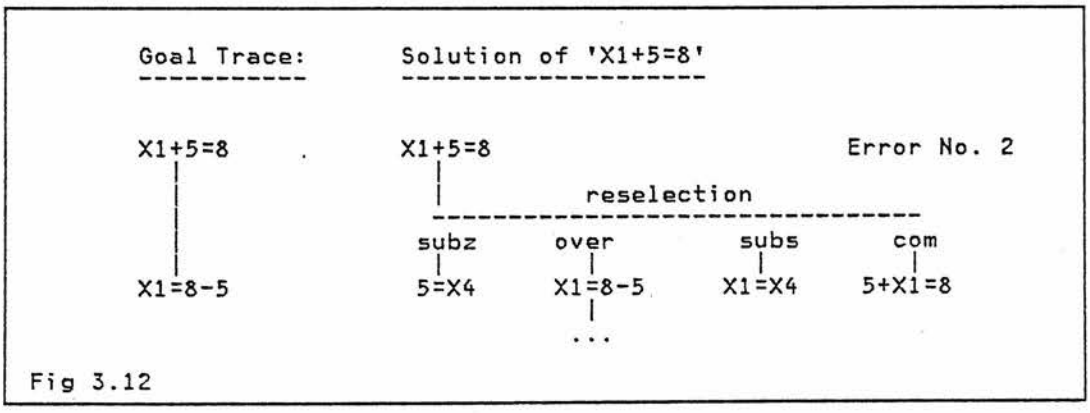


Fig 3.12

A special action is taken by the system with the aim of preventing the error from recurring again (the third time).

-*-

Why Do Errors Recur?

We have mentioned before that our system tries to resolve each conflict preferentially by modifying the existing constraints. New constraints are introduced only if the existing constraints cannot be modified (see section 7). Sometimes, however, constraints are wrongly deleted and this is one of the reasons why errors may recur.

Example

To correct the error shown in Fig. 3.11 clause 'over1' shown below was generated by our system. This clause is wrongly constrained since it can be selected only if 'X2' is 4.

```
over1 :   X1+X2=X3  <-  X2=:4                 & !  & X1=X3-X2
over2 :   X1+X2=X3  <-  var(X1) & int(X3)     & !  & X1=X3-X2
```

After some analysis we came to a conclusion that the system should have generated clause 'over2' which is shown in the previous figure. If no special action was taken clause 'over2' would never be generated irrespective of how many such errors were detected. To prevent this the system tries to recognize that a particular error has recurred and then takes a corrective action.

Detection of Recurrence of Errors

In our system recurrence of errors is detected by looking at the priority orderings. That is if a priority ordering Cj''> Ci is being introduced and if the priority ordering Cj'> Ci already exists, then this is taken as an indication that the current conflicting selection error has recurred. An action is taken to prevent the error from recurring again.

The context associated with the priority ordering Cj'> Ci is identified as an additional selection context. This context is used in the process of generating clause Cj''.

The additional selection contexts are used to 'prune out' the disjunctions of constraints generated on the basis of selection and rejection contexts S and R, as described in section 4. The predicates which are 'false' in the additional selection context(s) are simply deleted.

The use of additional selection context helps to generate constraints which are more 'general'. As the constraints generated are 'true' in two selection contexts the chances are quite high that they will be true in other contexts as well.

Example

Let us see how the second conflicting selection error shown in Fig. 3.12 was corrected by the system. The example is dealt with in more detail in section 14 where some of our experimental results are given.

The error mentioned was corrected by generation of clause 'over2'. The following contexts were used in the process:

Selection context S:

X1+5=8

Additional context:

X1+4=7

Rejection contexts:

R1: 3+2 = X1
R2: (3+1)+1 = X1
R3: 3+(1+1) = X1
R4: (X1+3)+1 = 7
R5: X1+(3+1) = 7

The following predicates could be used as constraints here:

Xi=:Xj, int(Xk) and var(Xl).

What these predicates mean is explained on page 108. Clause 'over2' was generated in several phases. In each phase one of the rejection contexts

Ri shown was considered together with the selection context S. In the first phase the rejection context R1 (3+2=X1) was used together with the selection context S (X1+5=8). The following intermediate version of clause 'over2' was generated as a result:

X1+X2=X3 <- (var(X1) v X2=:5 v int(X3)) & ! & X1=X3-X2

Next, the additional selection context 'X1+4=7' was used to 'prune out' the new disjunction of constraints. The constraint 'X2=:5' was found 'false' in this context and so it was deleted. The following clause was obtained:

X1+X2=X3 <- var(X1) v int(X3) & ! & X1=X3-X2

If the additional selection context was not used at this stage the constraint 'X2=:5' would have been retained and clause 'over2' would have been incorrectly constrained in the end.

Other similar modifications were performed when the other rejection contexts shown before were considered (eg. the predicate int(X3) was deleted etc.). The following clause was obtained in the end:

over2: X1+X2=X3 <- var(X1) & int(X2) & ! & X1=X3-X2

This clause is correctly constrained. It is the clause that we wanted the system to obtain.

-*-

Summary

If we want to prevent errors from recurring first we need to detect whether some particular error has recurred. Then we need to analyze the reason why this happened and finally take a corrective action to prevent this from happening again. This is also what our system does. However, further extensions could be made all three areas on the basis of further work.

3.11 LEARNING FROM EXAMPLES

Modifications of clauses need not only be instigated by errors. In our system clauses are sometimes modified after they have been used in new contexts. Errors are often prevented as a result, and so greater progress is achieved. The method of performing the modifications which is described in this section, will be referred to as the method of learning from examples.

Let us now see what the system does. First, the system checks if any of the clauses used in the search for a solution of the given problem can be modified. No modifications are performed, however, while the search continues. Relevant information is stored and the modifications are performed later, after the solution has been found.

Let us see how the system determines which clause is to be modified. After each clause has been selected, the system checks whether it contains any disjunctions of constraints. If it does it checks whether any of the disjuncts are 'false'. If any such disjuncts are found, the name of the clause is noted so that it could be modified later. The current selection context is also stored so that the system could determine which disjuncts were 'false'.

After the search has terminated, the information stored is retrieved and used as a basis for the modifications. First, the selection of the clause Ci stored is simulated. The context stored is used in the process. The head of the clause is matched against the particular goal stored as the context. If any constraint is found 'false' the corresponding constraint in the clause Ci is deleted.

Example

In this example we shall show how clause 'subs1' shown below is modified using the method described here. More details about how this clause was generated and when it was used are given in section 14.

```
subs1: X1+X2=X3  <- (X1=:(X5+X6) v X2=:1)  & !  & X1=X4  & X4+X2=X3
```

Suppose that the clause shown was selected in the following context: '(4+1)+2=X1'. One of the constraints of this clause is 'false' in this context and this is why the clause is modified by the system. The modifications are, however, performed only after the search has terminated.

To find how the clause 'subs1' is to be modified, the head of this clause is matched against the subgoal '(4+1)+2=X1' stored for this purpose. After the match the constraints of clause 'subs1' are as follows:

$$3+1=:(X5+X6) \quad v \quad 2=:1$$

We see that the predicate 2=:1 is 'false' and this is why the corresponding disjunct in the clause dealt with is deleted. The modified clause is as follows:

```
subs1: X1+X2=X3  <-  X1=:X5+X6 & !  & X1=X4  & X4+X2=X3
```

We notice that the new clause is a bit simpler than the original one. Also, the modifications like the one performed often prevent future errors. We shall discuss this point in more detail in the following.

Usefulness of Learning from Examples

When the method of learning from examples is employed errors are often avoided and so greater progress is achieved. To demonstrate this let us imagine that learning from examples is not used, and let us see how conflicting selection errors are corrected. Suppose that clauses Ci and Cj are to be selected in various contexts in the following order:

Cj, Ci, Ci, Cj.

The following figure shows how two conflicting selection errors would be corrected by the system under these circumstances.

Step	Clause Version to be Used	CSE Error	Resulting Priority Orderings
1	Cj	No	Cj > Ci
2	Ci	Yes	Ci'> Cj > Ci
3	Ci	No	Ci'> Cj > Ci
4	Cj	Yes	Cj'> Ci'> Cj > Ci

Fig. 3.14 Error Correction without 'Learning from Examples'

The conflict in step 2 would be resolved by generating clause Ci', a constrained version of clause Ci. New clauses generated by the system often contain one new disjunction of constraints. If by chance some of the disjuncts were 'true' in the selection context of clause Cj, an error would arise in step 4.

When the method of learning from examples is employed, the chances that the error in step 4 would occur are reduced. As the new selection context is encountered in step 3, clause Ci' is modified. Selection of this clause is, in effect, further constrained. It is much less likely that clause Ci' would be

incorrectly selected in step 4 and so clause Cj can be selected as a result.

Example

In our previous example we have shown how clause 'subs1' is modified by the system. This modification can, indeed, prevent an error, as we shall see.

Let us suppose that the following simple equation is to be solved: X1+1=4. This equation is normally transformed into the problem of subtraction (eg. X1=4-1). There is no point in trying to solve the subgoal 'X1=X4' which would be obtained if the clause 'subs1' was selected with the goal 'X1+1=4'. We notice that the original version of clause 'subs1' shown before can be selected with this goal. The modified version cannot be selected, however. So, our modification prevented the selection of the wrong clause in this context.

-*-

The number of errors which may be prevented by the method of 'learning from examples' depends on the sequence in which the problems are given. More errors are prevented if several problems of a similar kind are given before somewhat different problems. Clearly, the 'examples' have to be given so that 'learning from examples' could occur.

Should the method prevent, for example, the second conflicting selection error shown in Fig. 3.14, clause Ci has to be selected twice in succession before clause Cj is required.

A number of clauses were modified by the method of 'learning from examples' during our experiments. In section 14, for example, we show how clauses 'subs1', 'asod1' and 'subz1' are modified this way. Similar modifications were performed in

the experiments described in chapters 4.7 and 5.6. These modifications prevented many potential errors. The resulting clauses were also simpler.

Relationship to Conflict Resolution

The method of learning from examples complements the method of conflict resolution described before. The method of conflict resolution introduces new constraints which are intended to be as general as possible. When new examples are encountered and more information is available, the new constraints are specialized.

Extensions

The method of 'learning from examples' could be extended. As new selection contexts are encountered, the existing constraints could be replaced by other ones which are more specific provided they are 'true' in those contexts. The modification of the existing disjunctive constraints by deletion of some of the disjunct is just a special case of the more general method.

However, the process of replacement of constraints would have to be restricted in some way. If the existing constraints were replaced by more specific constraints several times the resulting clause could easily be overconstrained. This would have to be prevented. This could be done, for example, by restricting the actual number of attempts to replace the existing constraints by new ones. More work is needed, however, to develop these ideas further.

3.12 DEALING WITH DIFFERENT DOMAINS

We have mentioned that a certain number of clauses must be given to the the system so that it could learn to solve problems in a certain domain. It does not seem very practicable to require that all these clauses should be given to the system at the same time. If we had to do that then we would not be able to deal with any new problems from new domains, if the necessity arose. The only thing we could do is start right from the beginning which is rather cumbersome. The system should be able to accept new clauses at any time.

The system has to be careful, however. The clauses given cannot be simply added to the existing clauses because errors could occur with some of the old problems solved in the past. The clauses supplied could be wrongly selected in various contexts. To prevent this the new clauses have to be given appropriate priority. In the following we shall describe how these additional priority orderings are obtained in an automatic way.

The additional priority orderings are generated on the basis of the contexts associated with the existing priority orderings:

Context:	Ordering:
G	Ci > Cj

The system performs selection in the context 'G' and checks which clauses have been selected. If only clause Ci has been selected then no action is taken. If, however, some other clauses have been selected as well new orderings will be generated. Selection of clause Cx, for example, is prevented by the ordering 'Ci > Cx'.

The system goes through the orderings 'Ci > Cj' stored one by one, and in each case the associated context 'G' is

retrieved. Selection of each clause 'Cx' selected besides clause 'Ci' is prevented by the ordering 'Ci > Cx'.

Example

Suppose that the three clauses shown in the following figure have been given to the system initially and that the priority orderings shown have been generated by the system.

```
Given Clauses:
--------------
     subs1: X1+X2=X3  <-  X1=:(X5+X6) & !  & X1=X4   & X4+X2=X3
     subs:  X1+X2=X3  <-                !  & X1=X4   & X4+X2=X3
     subz:  X1+X2=X3  <-                !  & X2=X4   & X1+X4=X3

Priority Orderings:                Associated Contexts Stored:
-------------------                ---------------------------
     subs1 > subz                        (3+1)+1 = X1
     subz  > subs                         3+2 = X1
```

Suppose that we want the system to use the clause

asod: (X1+X2)+X3=X4 <- ! & X1+(X2+X3)=X4

and so we give it to the system. This clause, however, must be given an appropriate priority. The following priority ordering is, in fact, required: 'subs1 > asod'. This priority is also generated by the system to prevent selection of this clause in the context '(3+1)+1=X1' stored together with the ordering 'subs1 > subz'. The ordering 'subz > asod' is not needed, because clause 'asod' cannot be selected in the associated context.

Whenever our system is given any new clauses they are given appropriate priority in the way described. This happened several times in our experiments described in section 14. After the system has learnt to add integers it was given several new clauses so that it could deal with subtraction. When the system has learnt to do subtraction the system was given some more clauses useful for solving equations. In each step the clauses were given appropriate priority in the way described.

All existing clauses can be erased at any time together with all the associated priority orderings. This option is useful because it enables us to conduct a series of independent experiments, if we wish to.

3.13 IMPLEMENTATION

Our system was implemented in PROLOG running on the DEC-10 machines in Edinburgh and Dundee.

The problem solving subsystem, the error correction subsystem and the error detection subsystem described in this chapter were all implemented. The system implemented can correct both the simple selection errors and the conflicting selection errors. The approximate sizes of various subsystems are given in the following table. The numbers given show how many lines of code were used in the implementation excluding comments.

Size of Various Subsystems:

Overall Control	30
Problem Solving (control part)	35
Clause Selection	45
Error Detection	25
Error Correction (control part)	35
Conflict Resolution (control part)	25
Generation of Constraints	60
Modification of Constraints	20
Reorganization of Priority Orderings	20
Generation of Variable Instantiating Predicates *	50
Auxiliary Functions	40
Total	385

(*) This part of our system will be described in the next chapter.

3.14 EXPERIMENTAL RESULTS

Three sets of problems studied are shown here. The problems studied deal with addition of integers, subtraction of integers and simple equation solving. All three sets of problems are shown in the following figure. Other experimental results are given in chapters 4.7 and 5.6.

Addition	Subtraction	Equation Solving
* 3+2=X1	4-2=X1	* (X1+3)+1=7
* 4+3=X1	6-3=X1	* (X1+4)+1=8
(1+2)+(2+1)=X1	X1=7-4	* (2+1)+X1=5
((3+1)+(3+1))+2=X1		* (1+X1)+3=7
		(2+X1)+(3+1)=7
		((2+1)+2)+X1=9
		((3+X1)+2)+1=8

Fig. 3.16 Three Sets of Problems Used

In the following we shall describe how the problems shown were solved and which errors were detected in the course of their solution. Then we shall describe how these errors were corrected. The problems marked * in the previous figure will be dealt with in detail.

Types of Constraints Used

A set of predicates to be used as constraints was given to the system. The set given included the following predicates:

int(Xi) true, if Xi is an integer,
var(Xi) true, if Xi is a variable and
Xi=:Xj which is described below.

The predicate Xi=:Xj is true, if the terms Xi and Xj are identical, or

if they can be made identical by instantiating the variables in Xj. The term 'Xi' must be left unchanged. The following examples show when the predicate 'Xi=:Xj' is true, or false:

```
3+4   =: X1+X2    - true          X1+X2 =: 3+X2    - false
X3+X3 =: X1+X2    - true          X1+X2 =: 3+4     - false
3+X3  =: 3+X2     - true          X1+X2 =: X3+X3   - false
```

3.14.1 Addition of Integers

The initial set of clauses given is shown in Fig. 3.17. Clause 'asoc', for example, expresses associativity. No priority orderings were specified for the clauses given.

```
asoc:  X1+(X2+X3)=X4   <-    !  & (X1+X2)+X3=X4
subs:  X1+X2=X3    <-  !  & X1=X4  & X4+X2=X3
subz:  X1+X2=X3    <-  !  & X2=X4  & X1+X4=X3
eq:    X1=X1       <-  !
pred:  2 =X1       <-  !  & 1+1=X1
       3 =X1       <-  !  & 1+2=X1
       etc.
suc:   1+1 =X1     <-  !  & 2=X1
       2+1 =X1     <-  !  & 3=X1
       etc.
```

Fig. 3.17 The Initial Set of Clauses Given

Let us now see how the problem '3+2=X1' was solved by the system. The problem '3+2=X1' was given to the system as a goal and this initiated the search for the solution. Actually, a conjunction of goals was given to the system: '3+2=X1 & write(X1)'. The goal 'write(X1)' was used so that a valid comparison of 'results' could be made. The goal 'write(X1)'

may be interpreted as 'write out' the result obtained (X1). This predicate is 'true' by definition.

Fig. 3.18 shows the search tree obtained by our system together with the given goal trace.

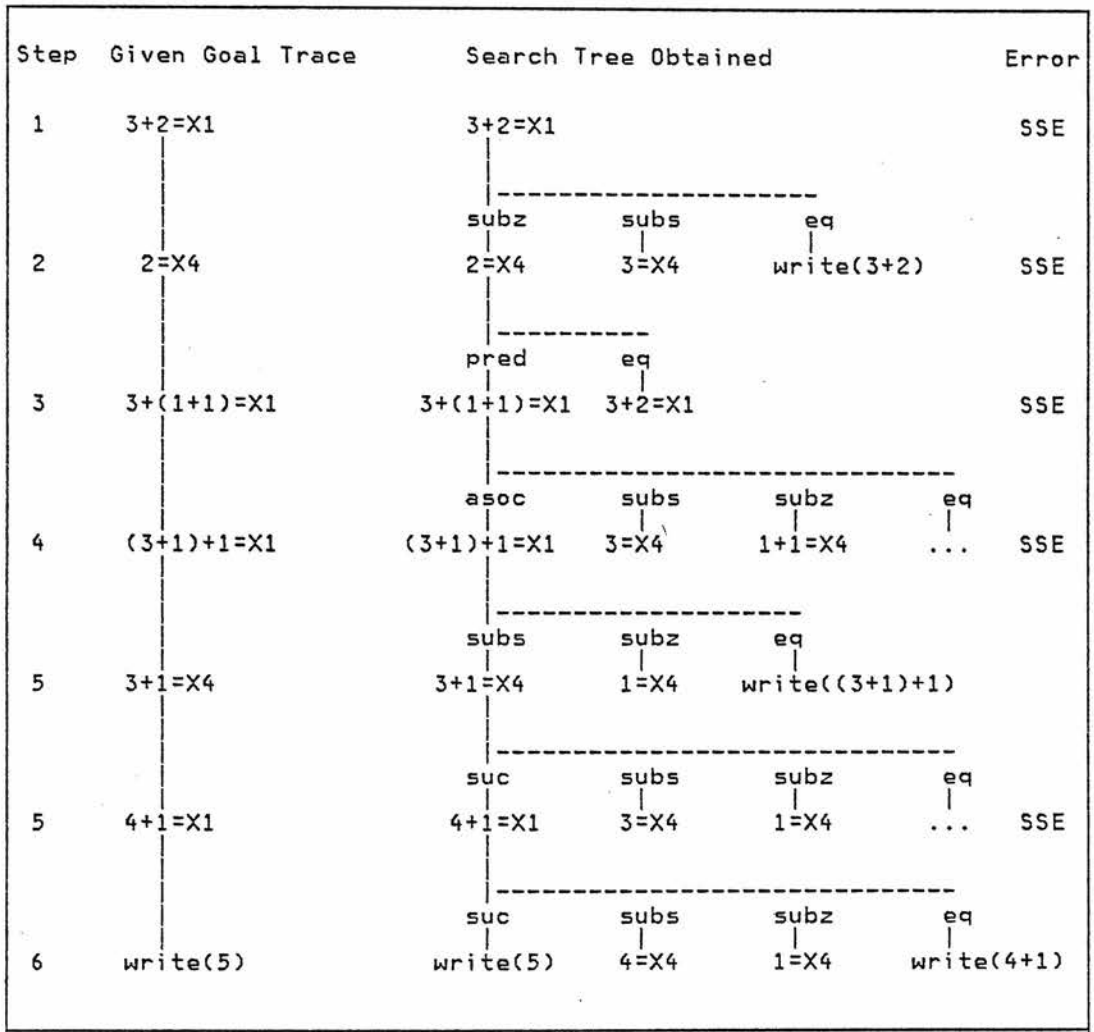


Fig. 3.18 Search for the Solution of 3+2=X1

Figure 3.19 shows the summary of the 'error information' stored during the search for the solution of this problem.

Step	Current Goal	Clause to be Selected	Type of Error
1	3+2=X1	subz	SSE
2	2=X4	pred	"
3	3+(1+1)=X1	asoc	"
4	(3+1)+1=X1	subs	"
5	3+1=X4	suc	"
6	4+1=X1	suc	"

Fig. 3.19 Error Information Stored

In the following we shall describe how the error information was used by the system to correct the errors detected.

Correction of Errors from Step 1.

The error detected in step 1 is a simple selection error. Clauses 'subz', 'subs' and 'eq' were all selected with the goal '3+2=X1'. What the system should have done is to select clause 'subz' only. This is what the error information stored shows. The error was corrected by introduction of two new priority orderings:

subz > subs and subz > eq.

Correction of Errors from Steps 2 and 3.

The errors detected in steps 2 and 3 were dealt with in a similar way as the previous error. The error detected in step 2 was corrected by the introduction of the priority ordering 'pred > eq'. The error detected in step 3 was corrected by the introduction of the priority ordering 'asoc > subz'.

The priority orderings 'asoc > subs' and 'asoc > eq' were not introduced. These priority orderings can be derived from the ones just added by transitivity. The system of priority orderings obtained is shown in the following figure:

Priority Orderings Obtained:	Associated Contexts Stored:
asoc > subz	3+(1+1)=X1
subz > subs	3+2=X1
subz > eq	3+2=X1
pred > eq	2=X2

Fig. 3.20

Correction of Errors from Step 4.

Originally, the error detected in step 4 was a simple selection error (see Fig. 3.19). However, because of the priority orderings added before, the error changed into a conflicting selection error. The system found this in the following way. It performed selection with the goal '(3+1)+1=X1' and found that the clause selected (subz) differed from the clause that should have been selected in this step (subs). The error information stored shows that clause 'subs' should have been selected here. To obtain the correct clause the system had to perform reselection. The following figure shows what happened.

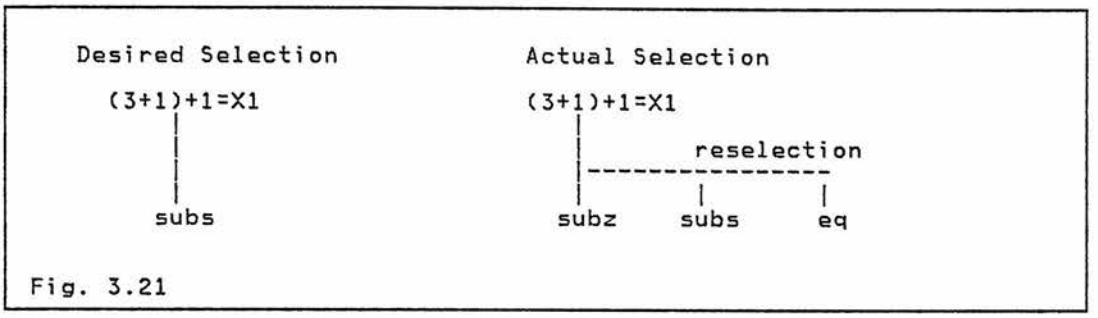


Fig. 3.21

The error detected in this step was corrected by generation of a constrained version of clause 'subs' called 'subs1'. This clause was given priority over clause 'subz'. The following contexts were used in the process of generating this clause:

Selection context: (3+1)+1=X1

Rejection context: 3+2=X1

The selection context was provided by the error detection subsystem. It is shown in Fig. 3.19 which shows the 'error information' stored. It is the context in which this error occurred.

The rejection context was identified in the following way. First, the system identified the priority ordering(s) which conflict the ordering 'subs > subz'. (This is the ordering that would be required if this error was corrected as a simple selection error.) Next, the associated context(s) were retrieved and used as the 'rejection contexts'.

The priority ordering 'subz > subs' was identified as the ordering which conflicts with 'subs > subz'. The associated context was 3+2=X1'. This context was used as the 'rejection context' by the system. To generate the new constraints both contexts were analyzed by the system. The example in section 4 showed how the constraints were actually obtained by the system. This is the new clause that was generated:

```
subs1: X1+X2=X3  <-  (X1=:(..+..)  v X2=:1)  & ! & X1=X4 & X4+X2=X3
```

Correction of Errors from Step 5.

The error detected in step 5 was corrected by addition of a priority ordering suc > subs1. The resulting system of priority orderings is shown in the following figure:

```
                asoc  > subz > subs
          suc > subs1 > subz >  eq
                        pred >  eq
```

Fig. 3.22 Priority Orderings Obtained

Other Problems of Addition

Let us now see how other problems of addition were solved with the system of clauses obtained. Fig. 3.23 shows how '4+3=X1' was solved. This problem was solved without errors. However, clause 'subs1' generated before was modified nevertheless. The clause was modified on the basis of 'learning from examples'. The method was described in section 11.

Step	Current Goal	Clause Selected
1	4+3=X1	subz
2	3=X4	pred
3	4+(1+2)=X1	asoc
4	(4+1)+2=X1	subs1
5	5+2=X1	subz
	...	...
11	write(7)	

Fig. 3.23 Another Problem of Addition Tried

When clause 'subs1' was selected in step 4 the system found that one of the existing constraints of clause 'subs1' was 'false' in the current

context. That is when clause

subs1: X1+X2=X3 <- (X1=:(..+..) v X2=:1) & ! & X1=X4 & X4+X2=X3

was matched against the goal (3+1)+2=X1, the constraint X2=:1 was instantiated to 2=:1. This constraint is 'false'.

After the problem '4+3=X1' was solved and the search has terminated clause 'subs1' was modified. First, the constraints which were 'false' in the current context were identified. Then the corresponding predicates in the clause 'subs1' were deleted. The new modified version of 'subs1' was as follows:

```
subs1: X1+X2=X3 <- X1=:(..+..) & ! & X1=X4 & X4+X2=X3
```

The next problem given to the system was (1+2)+(2+1)=X1. One simple selection error was detected in the course of the solution of this problem. This error was corrected by addition of

subs1 > asoc

to the existing system of priority orderings. The next problem given to the system was ((3+1)+(3+1))+2=X1. This problem was solved without errors.

What the System Has Learned

We believe that many similar problems can now be solved without errors. Our system has in fact learned how to add any number of integers. All the definitions of 'successors' and 'predecessors' of various integers used in these calculations must, however, be supplied to the system. (For example, '11' must be defined as the successor of '10' etc..)

We have mentioned that a number of clauses have been given to the

system initially. What the system has learned is how to use them in different situations. The action taken depends on what the expression dealt with looks like. The system has, in effect, learned the following rules.

When to Find the Successor of a Number

If you are trying to find the value of 'X1+1' find the successor of 'X1' but do not try to replace 'X1' by any other term. This is what is expressed by the clause 'suc' given to the system and the ordering 'suc > subs1' which has been acquired later.

How to Deal with (X5+X6)+X2

If the sum of 'X1' and 'X2' is to be worked out examine the term 'X1'. If 'X1' is term of the form 'X5+X6' calculate the sum of 'X5' and 'X6' but do not try to replace 'X2' by another term. This is expressed by clause 'subs1' which has been acquired by the system

```
subs1: X1+X2=X3 <- X1:=(X5+X6) & ! & X1=X4 & X4+X2=X3
```

and the ordering 'subs1 > subz'.

How to Deal with X1+X2

If the sum 'X1' and 'X2' is to be calculated and 'X1' is not a term of the form 'X5+X6' then try to replace 'X2' by another term. This is expressed by clause 'subz' which has been given to the system:

```
subz: X1+X2=X3 <- ! & X2=X4 & X1+X4=X3
```

Priority ordering subz > subs acquired ensures that 'X2' is dealt with next (but not 'X1'). If 'X2' is some integer it will be replaced by the term '1+X5' in the next step as a result of using clause 'pred' (predecessor).

How to Deal with X1+(X2+X3)

Any expression of the form X1+(X2+X3) should be transformed into (X1+X2)+X3. This is what is expressed by clause 'asoc' given to the system:

```
asoc: X1+(X2+X3)=X4 <- ! & (X1+X2)+X3=X4
```

Priority ordering asoc > subz which has been acquired by the system prevents the system from calculating the value of 'X2+X3'. This priority ordering ensures that the given integers are added from left to right.

3.14.2 Subtraction

The system learnt to subtract integers in a similar way to how it learned to add. The following problems were used in our experiments:

```
4-2=X1
6-3=X1
X1=7-4
```

Several clauses were added to the ones used before. These clauses enabled the system to find the solution of each problem given. They are shown in the following figure.

```
asom:  X1-(X2+X3)=X4   <-  !  & (X1-X2)-X3=X4
subm:  X1-X2=X3        <-  !  & X1=X4   & X4-X2=X3
subn:  X1-X2=X3        <-  !  & X2=X4   & X1-X4=X3
ovem:  X1-X2=X3        <-  !  & X2+X3=X1
come:  X1=X2           <-     var(X1) !  & X2=X1
```

Fig. 3.24 The Clauses Added

The following figure shows how the problem '4-2=X1' was solved by the system.

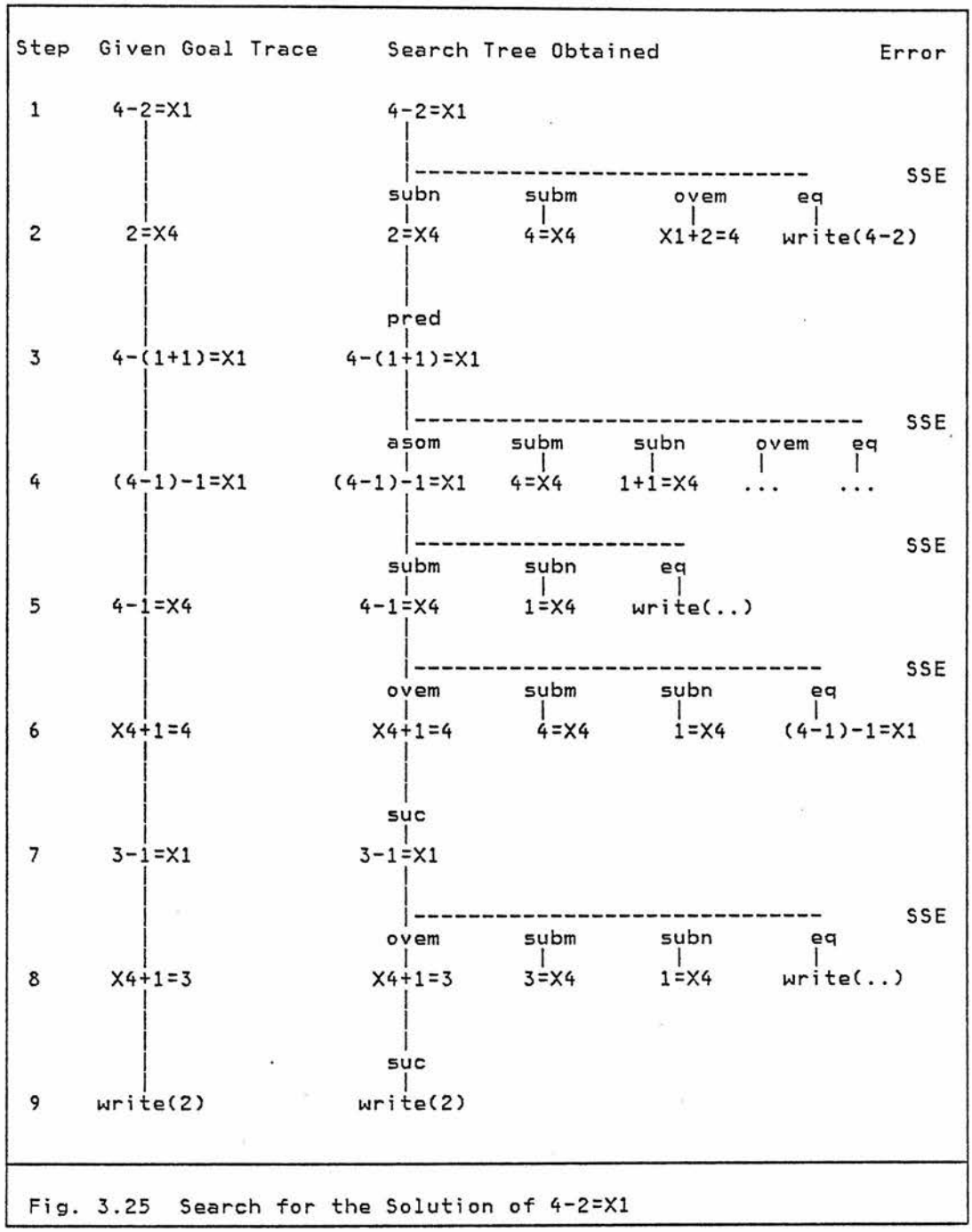


Fig. 3.25 Search for the Solution of 4-2=X1

The errors detected in steps 1 and 3 were dealt with as 'simple selection errors'. They were corrected by the introduction of new priority orderings. Because of the priority orderings added the errors detected in steps 4 and 5 were re-classified as 'conflicting selection errors'. They were corrected by conflict resolution. The new clauses

generated are shown in the following figure:

```
submi:  X1-X2=X3  <-  (X1=:(..-..) v X2=:1) & ! & X1=X4  & X4-X2=X3
ovemi:  X1-X2=X3  <-   int(X1)     & X2=:1  & ! & X2+X3=X1
```

Fig. 3.26 Clauses Generated by the System

The following figure shows the priority orderings obtained after all errors have been corrected:

```
                          subm > ovem
                  asom  > subn > subm
          ovemi > submi > subn >  eq
```

Fig. 3.27 Priority Orderings Obtained

3.14.3 Manipulation of Simple Equations

Our system can also learn to solve simple equations like these:

(X1+3)+1=7
(X1+4)+1=8
(2+1)+X1=5
(1+X1)+3=7
(2+X1)+(3+1)=7
((2+1)+2)+X1=9
((3+X1)+2)+1=8

The set of clauses obtained after the system has learned to add and subtract integers were used in our experiments dealing with the simple equation solving. Three other clauses which are shown in the following figure were additionally given to the system.

```
asod:  (X1+X2)+X3=X4   <-  !  & X1+(X2+X3)=X4
com:    X1+X2=X3       <-  !  &  X2+X1=X3
over:   X1+X2=X3       <-  !  &  X1=X3-X2
```

Fig. 3.28 Additional Clauses Used

The clauses added were given lower priority than some of the existing clauses. If this was not done errors would arise with some of the old problems of addition and/or subtraction. The priority orderings added are shown in the following figure together with the priority orderings generated by the system before.

```
suc > subs1 > asoc  > subz > subs      )   Priority orderings
                      subz >  eq       )   generated before
                      pred >  eq       )

      subs1 >                  asod    )    Priority
                      subz >   com     )    orderings
                      subz >   over    )    added
```

Fig. 3.29 Priority Orderings Used for this Example

The following figure shows how the first equation given was solved. The goal trace given helped the system to find the solution. In the first step, for example, clause 'subs1' was selected. With this clause, however, the solution cannot be obtained. Thanks to the goal trace given the error was discovered and reselection was performed. Reselection was performed similarly twice later.

Solution of the goal 'X1=7-4' is not shown in our figure since this subproblem can be solved without errors.

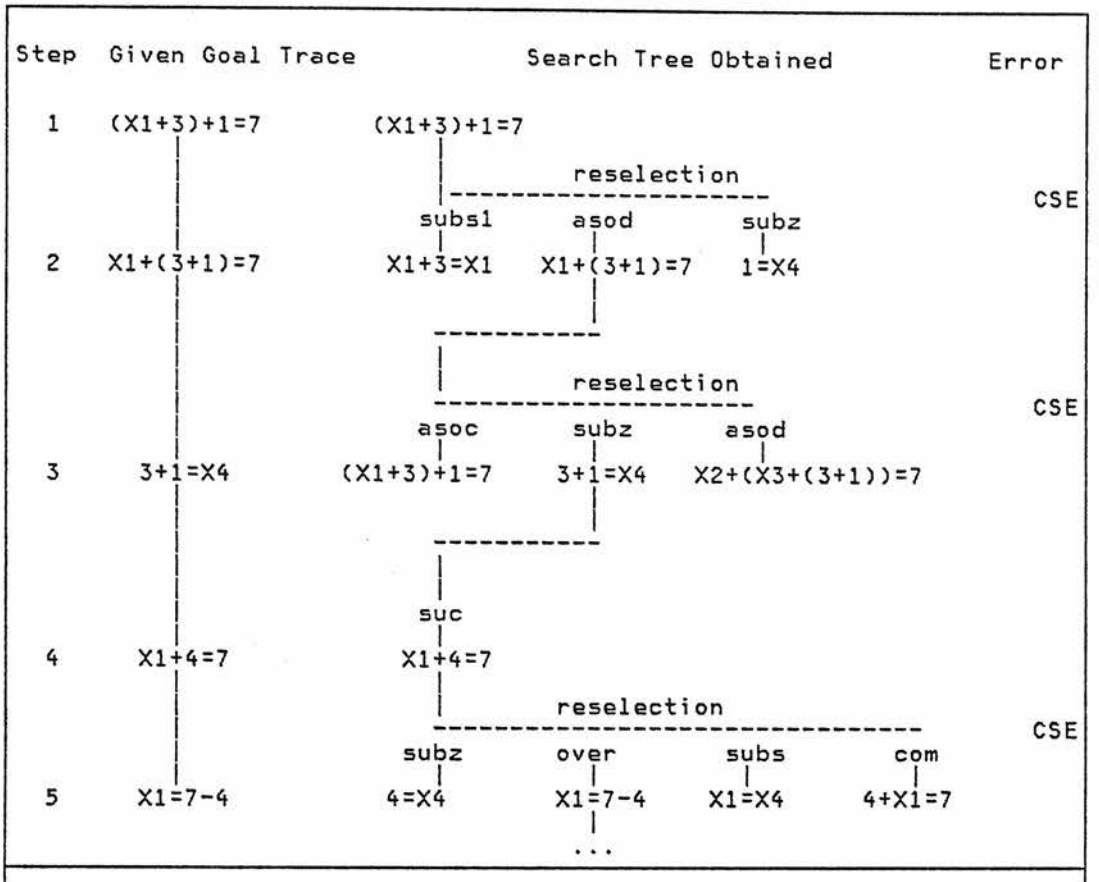


Fig. 3.30 Search for the Solution of (X1+3)+1=7

The following figure shows the 'error information' stored during the search for the solution of this problem. This information was used in the process of error correction which we shall describe in the following.

Step	Current Goal	Clause to be Selected	Type of Error
1	(X1+3)+1=7	asod	CSE
2	X1+(3+1)=7	subz	"
4	X1+4=7	over	"

Fig. 3.31 Error Information Stored

Correction of Errors from Step 1.

The first conflicting selection error arose because clause 'subs1' was selected instead of clause 'asod'. Had we tried to correct this error by addition of priority orderings only, a conflicting set of priority orderings would have been obtained (see the following figure).

```
suc > subs1 > asoc  > subz > subs     )
                      subz >  eq      )
                      pred >  eq      )   Current
                                      )   priority
       subs1 >                asod    )   orderings
                      subz >  com     )
                      subz >  over    )

                                      )   Conflicting
                 asod > subs1         )   priority ordering
```

Fig. 3.32 Current Priority Orderings

The error mentioned was corrected by conflict resolution. Clause 'asod1' was generated and given priority over clause 'subs1'. The following contexts were used in the process:

Selection context:	(X+3)+1=7	Assoc. Ordering:
Rejection context:	(3+1)+1=X1	subs1 > asod

The new clause generated is shown below. (The new set of priority orderings is shown in Fig. 3.33.)

```
asod1:  (X1+X2)+X3=X4   <-    ( var(X1) v X2=:3 v int(X4) )
                                 & !     & X1+(X2+X3)=X4
```

Correction of Error in Step 2.

The next conflicting selection error arose because clauses 'asod1' and 'asoc' were selected instead of clause 'subz'. Had we tried to correct this error by addition of priority orderings only, a conflicting set of priority orderings would have been obtained (see Fig. 3.33).

```
    suc > subs1 > asoc  > subz > subs      )
                          subz >  eq       )
                          pred >  eq       )   Current
                                           )   priority
 asod1 > subs1 >                  asod     )   orderings
                          subz >  com      )
                          subz >  over     )

                  subz > asod1             )   Conflicting
                  subz > asoc              )   priority orderings
```

Fig. 3.33 Current Priority Orderings

This conflicting selection error was corrected by generation of clause 'subz1' which is a new version of clause 'subz'. This clause was given priority over the clauses 'asod1' and 'asoc'. The following contexts were used in the process of generating this clause:

Selection context:	X1+(3+1)=7	Assoc. Ordering:
Rejection context:	3+(1+1)=X1	asoc > subz
- " -	(3+1)+1=X1	subs1 > subz
- " -	(X1+3)+1=7	asod1 > subs1

To generate the new clause each of the rejection contexts shown was considered together with the selection context. After the first rejection context (ie. 3+(1+1)=X1) was considered the following intermediate version of clause 'subz1' was generated:

```
X1+X2=X3  <-  ( var(X1)  v X2=:(3+..)  v int(X3) )
                 & !     & X2=X4      & X1+X4=X3
```

Next, the second rejection context (ie. (3+1)+1=X1) was considered. The clause generated before cannot be selected in this context and this is why the clause was not modified in this step. The constraints added in

the previous step prevent the selection of 'subz1' here.

As the third rejection context (ie. (X1+3)+1=7) was considered, the constraints generated before were modified again. The disjunct int(X3) which is 'true' in this context was deleted. Modification of constraints this way was described in section 7. The final clause version obtained was as follows:

```
subz1:  X1+X2=X3   <-    ( var(X1)  v X2=:(3+..) )
                            & !     & X2=X4     & X1+X4=X3
```

The new set of priority orderings is shown in the following figure.

Correction of Errors from Step 4.

The next conflicting selection error arose because clause 'subz1' was selected instead of clause 'over'. Had we tried to correct this error by addition of priority orderings only, a conflicting set of priority orderings would have been obtained as the following figure shows:

```
          subz1 >          asoc  > subz > subs      )
           suc  > subs1 > asoc  > subz >  eq       )
                                pred >  eq         )   Current
                                                   )   priority
 subz1 > asod1 > subs1 >                  asod     )   orderings
                                subz >    com      )
                                subz >    over     )
                                                   )
                                                   )   Conflicting
                           over > subz1            )   orderings
```

Fig. 3.34 Current Priority Orderings

The conflicting selection error was corrected by generating a new version of clause 'over', called 'over1'. This clause was given priority over clause 'subz1'. The following contexts were used in the process:

Selection context: X1+4=7

		Assoc. Ordering:
Rejection context:	3+2=X1	subz > over
- " -	(3+1)+1=X1	subs1 > subz
- " -	3+(1+1)=X1	asoc > subz
- " -	(X1+3)+1=7	asod1 > subs1
- " -	X1+(3+1)=7	subz1 > asoc

Each of the rejection contexts shown was used in conjunction with the selection context in the way described before. When the first rejection context (ie. 3+2=X1) was considered the following clause was generated:

X1+X2=X3 <- (var(X1) v X2=:4 v int(X3)) & ! & X1=X3-X2

This clause was modified again when the other rejection contexts were considered. The new constraints should be 'false' in all the rejection contexts. The following clause was obtained in the end:

over1: X1+X2=X3 <- X2=:4 & ! & X1=X3-X2

This clause, however, is incorrectly constrained, since it can be selected only when 'X2' is 4. The constraint 'X4=:4' was chosen, because it was the only constraint which is 'false' in all the rejection contexts shown. What the system should have done is to generate the following conjunction of constraints:

var(X1) & int(X2).

This conjunction is also 'false' in all the rejection contexts used here. Clause 'over2' which is generated later is, however, correctly constrained. It contains just this conjunction of constraints. A special action is taken by the system to generate this clause, as we shall see.

The following figure shows the priority orderings as they were after the last error was corrected:

```
                  subz1 >          asoc  > subz > subs
                   suc  > subs1 > asoc  > subz >  eq
                                          pred >  eq

 over1 > subz1 > asod1 > subs1 >                  >  asod
                                          subz >  com
                                          subz >  over
```

Fig. 3.35 Current Priority Orderings

Another Problem Tried

The next problem dealt with was chosen to be similar to the previous problem. Only one error was detected in the course of the solution of this problem, as the following figure shows. The error arose because clause 'over1' was incorrectly constrained before.

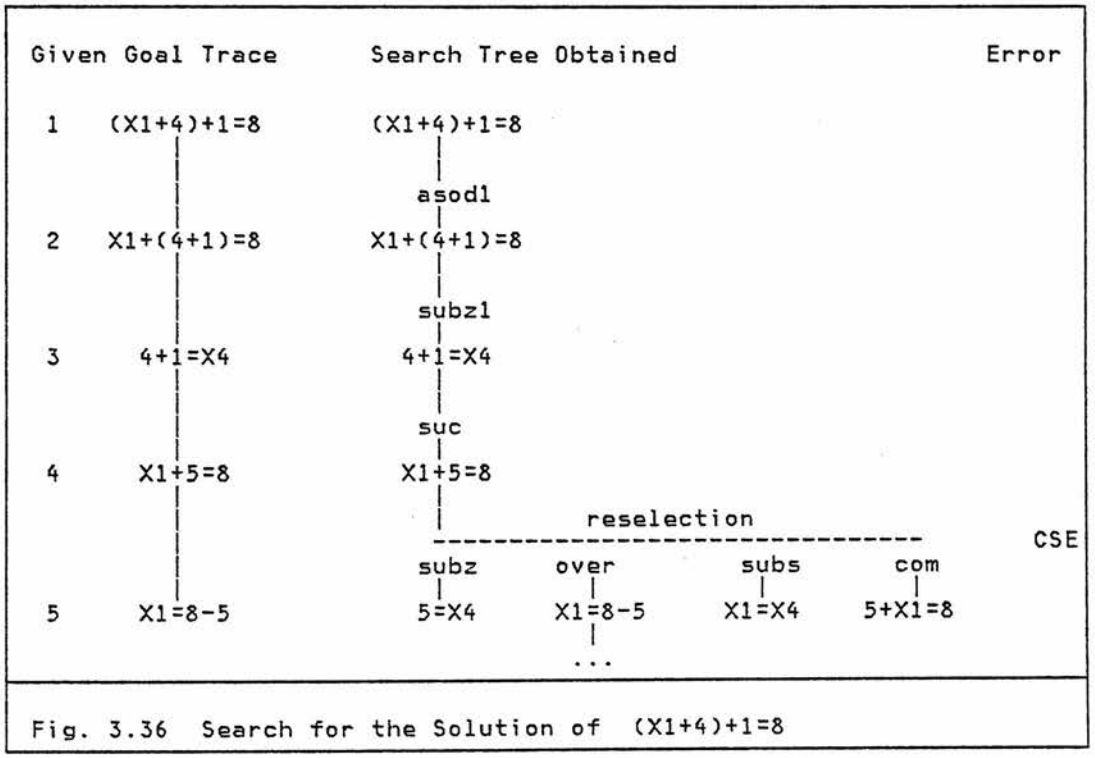


Fig. 3.36 Search for the Solution of (X1+4)+1=8

A special action was taken by the system with the aim of preventing a repeated recurrence of this error. The method was described in section 10. The system identified a particular context as an 'additional

selection context' and used it in the process of generating clause 'over2'. The 'additional selection context' is shown in the following together with various other contexts used here. We notice that the rejection contexts used here are identical to the rejection contexts used before, when clause 'over1' was generated.

Selection context:	X1+5=8	
Additional context:	X1+4=7	
		Assoc. Ordering:
Rejection Context:	3+2=X1	subz > over
- " -	(3+1)+1=X1	subs1 > subz
- " -	3+(1+1)=X1	asoc > subz
- " -	(X1+3)+1=7	asod1 > subs1
- " -	X1+(3+1)=7	subz1 > asoc

The following version of clause 'over2' was generated on the basis of the selection context 'X1+5=8' and the rejection context '3+2=X1':

X1+X2=X3 <- (var(X1) v X2=:5 v int(X3)) & ! & X1=X3-X2

Next, the additional selection context 'X1+4=7' was used to prune down some of the disjuncts. The constraint 'X2=:5' was found 'false' in this context and so it was deleted. The following clause was obtained as a result:

X1+X2=X3 <- var(X1) v int(X3) & ! & X1=X3-X2

If the additional selection context was not used at this stage the constraint 'X2=:5' would have been retained and the resulting clause would have been incorrectly constrained again.

The intermediate version above was modified again when the other rejection contexts were considered. The clause finally obtained was as follows:

over2: X1+X2=X3 <- var(X1) & int(X2) & ! & X1=X3-X2

We see that the constraints of this clause are more general than the constraints of the clause 'over1' which was generated before.

The priority orderings obtained were as follows:

```
                                subz1 >          asoc  > subz > subs
                                 suc  > subs1 > asoc  > subz >  eq
                                                      pred >  eq

over1 > over2 > subz1 > asod1 > subs1 >                       asod
                                                      subz >  com
                                                      subz >  over
```

Fig. 3.37 Current Priority Orderings

Two of the existing clauses were also modified on the basis of the method of 'learning from examples' described in section 11. Both clauses were selected during the solution of the equation dealt with last ((X1+4)+1=8). The modified clauses were as follows:

```
asod1:  (X1+X2)+X3=X4 <-  (var(X1) v int(X4))  & !  & X1+(X2+X3)=X4
subz1:  X1+X2=X3      <-   var(X1) & !  & X2=X4     & X1+X4=X3
```

Other Equations Solved

Two other problems given to the system were: '(2+1)+X1=5' and '(1+X1)+3=7'. The following figure shows how the first problem was solved with the existing clauses. Solution of the subgoal 'X1=5-3' is not shown since this problem can be solved without difficulties. We see that one conflicting selection error was detected in the course of the solution.

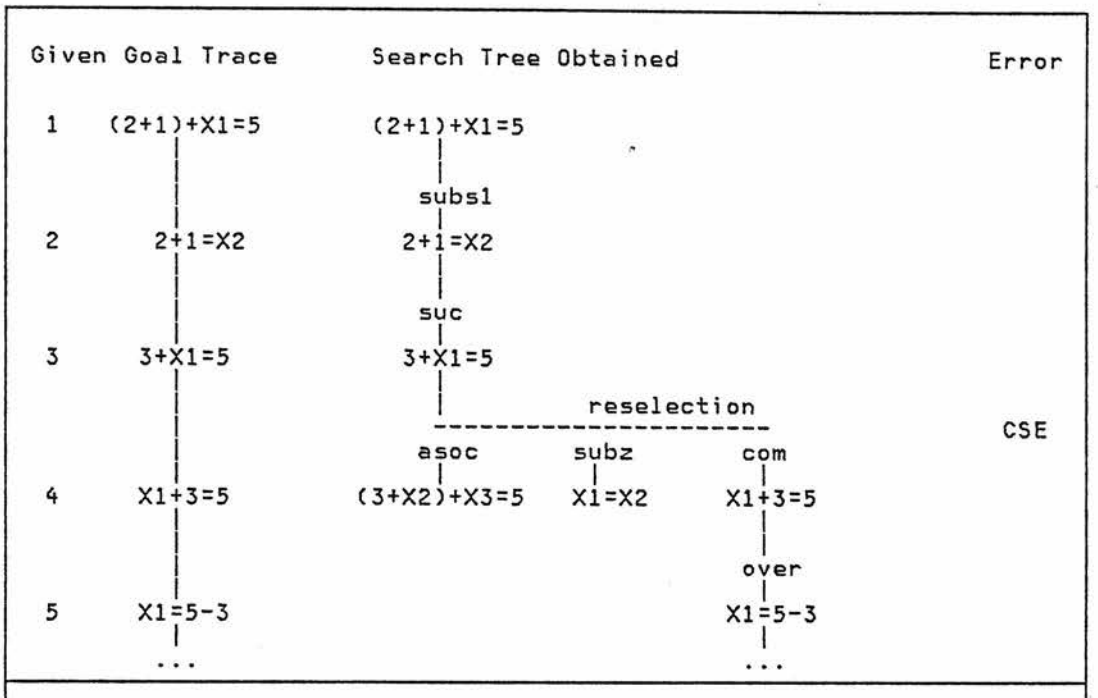


Fig. 3.38 Search for the Solution of (2+1)+X1=5

The conflicting selection error detected was corrected by conflict resolution. The new clause generated from clause 'com' was called 'com1'. The new clause generated was as follows:

```
com1:   X1+X2=X3   <-  (var(X2) v int(X3))  & !  & X2+X1=X3
```

The new clause was given priority over clause 'asoc'. The system of priority orderings obtained was as follows:

```
                    subz1 >                          asoc  > subz > subs
                            suc   > subs1 > asoc  > subz >  eq
                                    com1  > asoc
                                                     pred >  eq
  over1 > over2 > subz1 > asod1 > subs1 >                          asod
                                                     subz >  com
                                                     subz >  over
```

Fig. 3.39 Current Priority Orderings

The following figure shows how the next problem was solved. We see that the given problem is transformed into the following problem: (X1+1)+3=7. This problem can be solved without difficulties since a similar problem was dealt with before ((X1+3)+1=7).

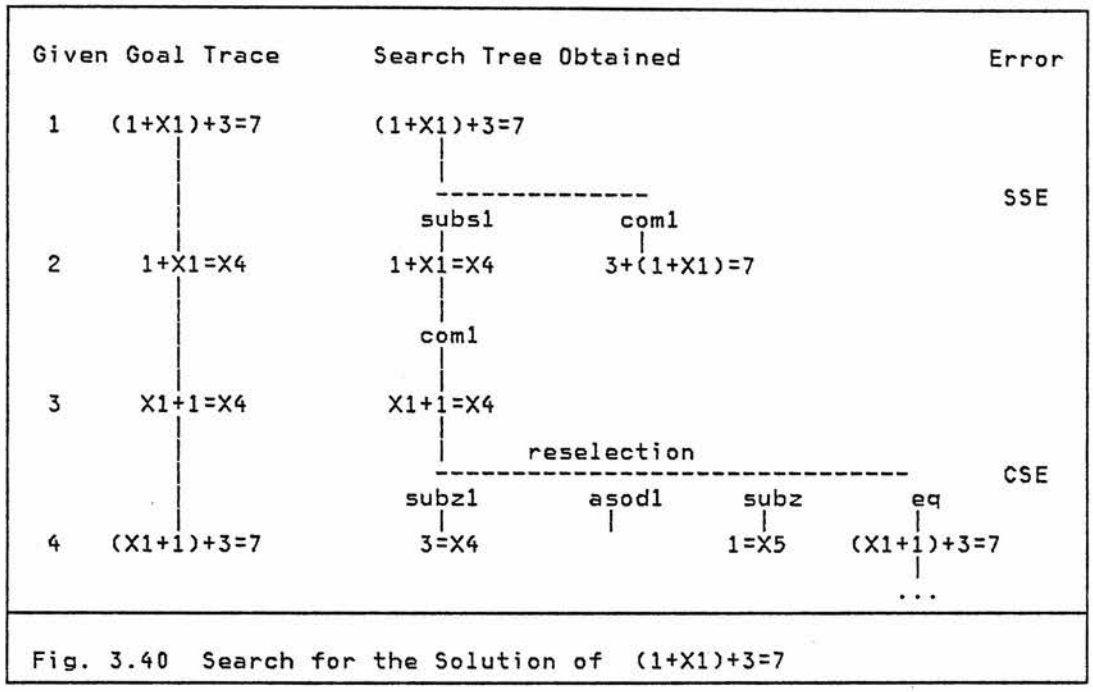


Fig. 3.40 Search for the Solution of (1+X1)+3=7

The simple selection error detected in step 1 was corrected by addition of the following priority ordering:

subs1 > com1.

The conflicting selection error detected in step 3 was corrected by conflict resolution. Clause 'eq' was constrained and the new clause version 'eq1' obtained was given priority over clause 'subz1'. The new clause generated was as follows:

```
eq1:  X1=X1 <-  X1=:X2+X3 & var(X2) & !
```

The system of priority orderings obtained was as follows:

```
          eq1  > subz1 >                                  asoc  > subz > subs
                                                       > asoc  > subz >  eq
                          suc  > subs1 > com1          > asoc
                                                         pred >  eq
 over1 > over2 > subz1 > asod1 > subs1                 > asod
                                                         subz > com
                                                         subz > over
```

Fig. 3.41 Priority Orderings Obtained

What the System Has Learned

The system is now capable of solving many other equations similar to the ones solved, provided all the terms consist of the function symbol '+', any number of integers and one variable. The system has a somewhat more limited capability of dealing with equations with '-'.

The system is capable of handling the given equations in a similar way as we do it. It is capable of isolating the unknown by transferring various terms to the other side of the equation. It is capable of evaluating various subterms which makes the equation dealt with simpler. It is capable of using associativity and commutativity rules in the right situations. More details about what the system can do are given in the following.

Isolation of Variables

If the equation dealt with contains a term with a variable and an integer then try to isolate the variable by transferring the integer to the other side to the equation. The equation can then be solved without difficulties (using subtraction). This is expressed by clause 'over2' which has been acquired by the system:

```
over2: X1+X2=X3 <- var(X1) & int(X2) & ! & X1=X3-X2.
```

Evaluation of Subterms

If it is not possible to isolate the variable in the way just described then try to examine the term 'X1+X2' on the left hand side of the equation. If 'X1' is a variable try to evaluate the subterm 'X2'. This is expressed by clause 'subz1' which has been acquired the system

```
subz1: X1+X2=X3 <- var(X1) & ! & X2=X4 & X1+X4=X3
```

and the ordering 'over2 > subz1' which gives priority to clause the 'over2' shown before.

If it is possible to use clause 'subz1' there is no point in applying the associativity rule. This is expressed by the orderings

```
subz1 > asod1
subz1 > asoc
```

which have been acquired by the system.

Use of Associativity

The associativity rule may be used to rearrange the left hand side of the equation dealt with, if it is of the form '(X1+X2)+X3' and if 'X1' is a variable. Use of this rule will help to isolate the variable 'X1'. The following clause expresses what we have just said:

```
asod1: (X1+X2)+X3=X4 <- var(X1) v int(X4) & ! & X1+(X2+X3)=X4.
```

We notice that the clause above contains an additional condition 'int(X4)' which does not seem to have any intuitive meaning. It tells us that clause 'asod1' can be selected if the right hand side of the equation is an integer.

If the rule just mentioned can be used one should not try to evaluate the term 'X1+X2'. This is what the priority ordering 'asod1 > subs1' acquired by the system tells us.

This is quite a sensible restriction, really. If 'X1' happens to be a variable, preference is given to clause 'asod1' over clause 'subs1', and so the system will not try to evaluate 'X1+X2' which would not have any sense.

Use of Commutativity

If the associativity rule mentioned cannot be applied the commutativity rule may be used to interchange the subterms in 'X1+X2', provided 'X2' is a variable. This is what clause 'com1' expresses:

```
com1: X1+X2=X3 <- (var(X2) v int(X3)) & ! & X2+X1=X3.
```

If 'X1' is a term of the form 'X5+X6' the addition of 'X5' and 'X6' should be performed. The ordering subs1 > com1 forces us to do that.

Notice that we were dealing with the equation '3+X1=5', for example, and clause

```
asoc: X1+(X2+X3)=X4 <- & ! & (X1+X2)+X3=X4
```

was applied, the equation would have been transformed into '(3+X2)+X3=5'. This does not seem desirable, because the equation obtained contains two variables instead of one. This, however, is prevented if we respect the ordering 'com > asoc' which has been generated by the system.

Use of Substitution

If both sides of the equation dealt with can be matched and if the left hand side of equation is in a suitable form (ie. it is a term of the form 'X2+X3' and 'X2' is a variable) the equation should be regarded as solved and attention should be paid to other outstanding goals. This will be achieved whenever clause 'eq1' will be applied:

```
eq1:  X1=X1 <- X1=:X2+X3 & var(X2) & !.
```

Notice that as the two sides of the equation are matched substitutions may occur in the other equations which are awaiting solution.

If the rule just mentioned can be applied and the left hand side of the equation is a term of the form 'X2+X3' one should not try to replace the subterm 'X3' by any other term. The priority ordering 'eq1 > subz1' forces us to do that.

Dealing with Other Types of Equations

Our system could be taught to solve many other types of equations which we have not shown here. For example, it could be taught to solve equations containing various terms with + and - such as this one:

```
(5+X1)-1=2.
```

The system described would, however, not be able to solve this equation:

```
3*(X1-2)=9.
```

The system described (ELM1) cannot solve this equation, because it does not know how to solve one of the subproblems - that is how divide 9 by 3, and it could never learn to do it (*). Our extended system ELM2, however, can learn to do division without difficulties, and so it can also learn to solve the equation shown above.

3.15 DISCUSSION

In this chapter we have described how selection errors are corrected by the system. Simple selection errors are corrected by addition of new priority orderings only. The method was described in section 3. Conflicting selection errors are more difficult to correct. Each error is corrected by rearranging the existing priority orderings and by modification of one of the existing clauses. The method was described in section 4 and the following sections.

In both cases the priority orderings play an important role. Use of explicit priority orderings has certain advantages over other approaches. In the following we shall explain why we used them.

Use of Priority Orderings

We have mentioned that simple selection errors are corrected by introduction of new priority orderings for the existing clauses. We could have put the existing clauses into a list and then tried to eliminate these errors by reordering

(*) More precisely ELM1 cannot learn the method of division described in chapter 5.

the clauses in this list. Use of explicit priority orderings has the following advantages.

Firstly, there is no danger that clauses could be repeatedly reordered.

Secondly, the use of priority orderings facilitates the correction of conflicting selection errors. The clauses whose selection should not be prevented can be easily identified in the way described in section 8. The selection contexts of these clauses are used as 'rejection contexts' in the process of generating the new constrained clause. If other contexts were taken into account as well the resulting clause could easily become overconstrained.

Example

Suppose a conflicting selection error arose because a priority ordering Ci>Cj prevented selection of clause Cj. In our system the conflict is resolved by generation of clause Cj' and addition of Cj'>Ci. The selection context of clause Ci is used by the system in the process of generating the constraints of clause Cj'.

Suppose the clauses Ci and Cj appeared in a list together with some other clause Ck. If the clauses appeared in the order Ci,Ck,Cj we would in effect have Ci>Ck>Cj. The selection context of clause Ck would be also be taken into account (unnecessarily) in the process of generating the clause Cj'. The new constraints would have to prevent the selection of clause Cj' in this context and clause Cj' could be overconstrained as a result.

Use of Contexts

New constraints are generated on the basis of 'selection'

and 'rejection' contexts. Basically, the system looks for predicates which would differentiate between the two types of contexts. In this respect our method is similar to the method described by Winston (1970).

There is no need, however, that the examples of contexts (Winston's concepts) are manually provided. The selection and rejection contexts are obtained automatically by the system. How this is done was explained in section 4.

We notice that a number of 'rejection contexts' may be used in the process. It is as if the descriptions of a 'house', 'tent' etc. were used as 'negative examples' in the process of refining the concept of an 'arch'. Winston (1970) did not use this information in any way. A more detailed explanation of how our work relates to Winston's can be found in chapter 6.

Choice of Constraints

New constraints are obtained by scanning the given set of predicates and selecting those that are true in the selection context and false in the rejection context. If several constraints are found, they are all used as disjuncts in the final expression generated.

The system looks for constraints which are as general as possible. More general constraints are preferred in order to minimize the chances of generating overconstrained clauses.

If other selection errors occur the clauses dealt with before may be modified again. The system tries to modify the existing constraints first, but if this cannot be done new constraints are added to the existing ones. Thus clauses in our system will get more and more constrained as a result of error correction. It is not possible for the system to loop,

that is to add new constraints, then delete them and so on.

More work is needed in this area, however. We have mentioned that our aim is to generate constraints which are as general as possible. This is not always achieved. We have explained this in section 6. More work is needed to establish how the situation could be improved.

We think that the system should be able to replace the existing constraints by other more suitable ones later, when more information is available. Perhaps further progress could be achieved this way.

Also, the use of more complex kinds of constraints should be investigated. A number of clauses could be used to define the meaning of such constraints. Recursive definitions could be used, too.

The definitions of constraints need not be correct either. More work is needed to establish how the errors caused by that should be detected and corrected.

Role of Learning from Examples

Modifications of clauses need not only be instigated by errors. In our system modifications are sometimes performed after the new clauses have been selected and applied in new contexts. The modification performed often prevent further errors, and so a better progress is achieved.

The modifications performed in our system as a result of 'learning from examples' are limited: the existing disjunctions of constraints may be modified by deleting some of the disjuncts. The new selection contexts encountered are used to identify the disjuncts that could be deleted. The

details were given in section 11.

The method of 'learning from examples' supplements the method of 'learning on the basis of errors'. We have mentioned that the aim of conflict resolution is to generate constraints which are as general as possible. The constraints are under certain circumstances specialized on the basis of 'learning from examples'.

The method of learning from examples could be extended. The constraints introduced by conflict resolution could be replaced by other constraints which are more specific even though great care would have to be taken. The process of replacement would have to be restricted in some way. If this was not done the resulting clauses could easily be overspecialized.

Preventing Recurrence of Errors

The clauses generated by the system are sometimes not correctly constrained. It is unlikely that in any system the clauses would always be correctly constrained since when errors are being corrected, only a limited amount of information is used (ie. particular selection and rejection contexts). This, we believe, has to be accepted. However, the system ought to recognize that clauses might have been incorrectly constrained and then take some action to avoid further problems. This is what our system does. It tries to use more than one selection context in the process of generating the new clause.

We believe that further extentions in this area could be made bearing in mind that the system should try to assess whether the modifications performed before have been

beneficial or not. Also, it is important that more information is given to the error correction subsystem so that it could come up with a better modification than the ones performed in the past. We believe that it would be worth while studying how these principles might be best exploited.

4 CORRECTION OF INSTANTIATION ERRORS

4.1 INTRODUCTION

In this chapter we discuss how instantiation errors are dealt with by the system. We show how instantiation errors are detected, and how they are corrected by modification of the existing clauses. Examples are given illustrating how the techniques described are applied.

Detection of Instantiation Errors

First, we discuss how instantiation errors are detected by the system and what error information is stored when each error is detected. This includes the name of the faulty clause to be modified, and clause instances on the basis of which the modifications are performed (*). An explanation is given of how this information is obtained.

Detection of instantiation errors is discussed in section 2 in detail. A brief explanation of this topic was given in chapter 2.6.

(*) A clause instance is a clause instantiated in a particular way.

Correction of Instantiation Errors

Correction of all types of errors is performed only after the search has terminated. Correction of instantiation errors is performed on the basis of the error information stored. The clause instances stored are analysed together with the 'faulty clause'. The aim of this analysis is to identify the variables in the faulty clause and the corresponding subterms in the clause instances used. Then various predicates from a given set are tried out to see if they can be used as the new variable instantiating predicates. The predicates found are used to modify the 'faulty clause'. The details of this are given in section 3.

Elimination of Additional Selection Errors

If both selection and instantiation errors have been detected with some particular goal, both errors are corrected by modification of one clause. The instantiation error is corrected by the addition of variable instantiating predicates. The selection error is corrected by the addition of priority orderings and/or predicate constraints.

New predicates are always added to the left of the existing predicates, irrespective of whether they are constraints, or variable instantiating predicates.

Some Problems with Variables

Certain type of instantiation errors cannot be detected by the system, as it is implemented. The system cannot detect if one and the same variable is used in different nodes in the search tree. This problem is discussed in section 5.

The problem of detection arises because of the way the search has been implemented. We describe how the system could be extended so that it could detect the errors mentioned. We describe a technique of freezing instantiations in the term specified and explain how the problems mentioned can be overcome with the help of this technique.

Experimental Results

Examples are given illustrating how the techniques described are applied. We show how problems are solved, how errors are detected and how they are corrected. Both instantiation and selection errors are dealt with. The examples presented here are from the domain of letter series completion. The techniques described can equally well be applied in the domain of arithmetic and algebra.

4.2 DETECTION OF INSTANTIATION ERRORS

We have mentioned before that errors in our system are detected on the basis of comparison of goals. The goals in the given goal trace are compared with the corresponding goals in the given trace. If the goals obtained by the system are not instantiated as the goals in the goal trace an instantiation error is detected.

We assume that if the error was not detected the solution obtained by the system would be too general. For example, a variable could be returned as the solution of the problem of addition of two integers. This is not acceptable.

Example

The following figure shows an example of an instantiation error.

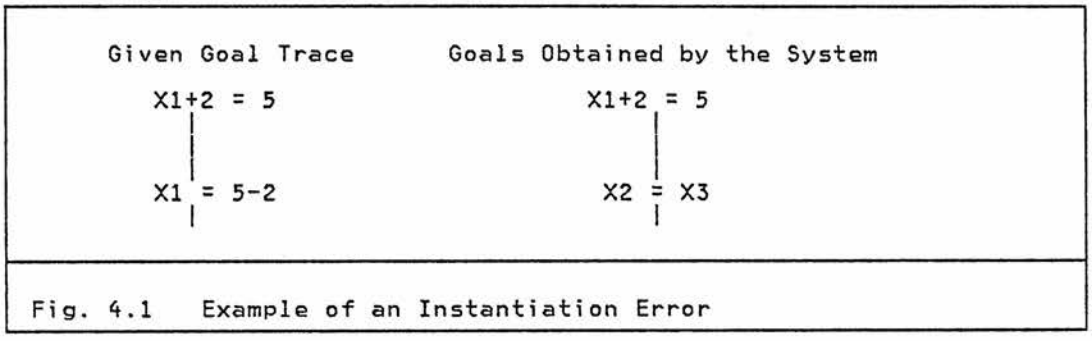


Fig. 4.1 Example of an Instantiation Error

We see that the goal 'X2=X3' which has been obtained by the system is an uninstantiated version of the corresponding goal in the given goal trace. It is more 'general' than the corresponding goal in the goal trace. This is why an instantiation error is detected here.

-*-

To detect instantiation errors we could consider various subterms in Gs, the goals obtained by the system together with the corresponding subterms in Gt, the goals from the given goal trace. If the subterm of Gs was a variable and the corresponding subterm of Gt some constant (or a term), an instantiation error could be detected.

Unfortunately, however, if this method was used certain types of errors could not be detected. An example of an error which could not be detected is shown in Fig. 4.2. The goals obtained by the system (Gs) contain several distinct variables (X1,X2) instead of several occurrences of one variable (X1).

In order to detect all types of instantiation errors the following method is used. The system

- checks if Gs and Gt match,
- replaces all distinct variables in Gs by constants,
- checks again if Gs and Gt match.

Replacement of variables in Gs by constants will prevent further matching. So, if the goals do not match, instantiation error is detected.

Example

Let us consider the goals in the following figure. Let us see if the system can detect that an instantiation error has occurred here.

Step	Given Goal Trace	Goals Obtained by the System
1	2-X1 = X1	2-X1 = X1
2	2 = X1+X1	2 = X1+X2

Fig. 4.2 Another Example of an Instantiation Error

First, the system tries to match the goals obtained in step 2. We see that these goals match. Next, the system replaces the variables obtained by the system (2=X1+X2) by distinct constants (eg. q1,q2). The goal '2=q1+q2' is obtained as a result. However, this goal does not match the corresponding goal in the goal trace (2=X1+X1) any more and this indicates that an instantiation error has occurred.

-*-

Our system cannot detect one particular type of an instantiation error. It cannot detect if one and the same variable is used in different nodes in the search tree. An

example of such an error is shown in the following figure.

Example

The goals obtained by the system contain two distinct variables (X1,X2). These goals belong to two different 'nodes' This is incorrect, because the goals in the goal trace contain only one variable (X1).

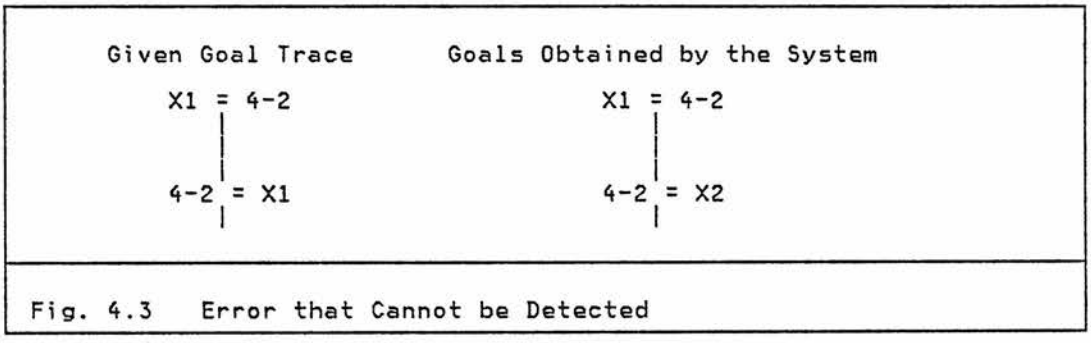


Fig. 4.3 Error that Cannot be Detected

In section 5 we shall describe how the system could be extended so that this type of error could be detected as well.

Error Information Stored

Whenever an instantiation error is detected the following error information is stored:

- the type of the error (ie. instantiation error),
- C - the name of the 'faulty clause',
- Cf - the faulty clause instance,
- Cd - the desired clause instance.

The 'faulty clause' is the clause that needs modifying. To find out which clause is the 'faulty clause' the system tries to determine which clause has introduced the current subgoal. This clause is then regarded as the 'faulty clause'. We have assumed that each error can be corrected by modification of

one clause only.

Let us see how the clause instances are obtained. As clause C

G <- G1 .. Gn

is applied, the head of the clause is matched against the current goal. The predicates in the body of clause C are used as the new 'goals'. All of these goals are stored on the goal stack. Each goal is marked off after it has been solved.

If an instantiation error has been detected while the system was trying to solve one of the goals 'G1..Gn' the current clause instance can be obtained without difficulties. It is simply assembled from the goals stored. This clause instance obtained is used as the faulty clause instance Cf. This clause instance may differ from the original clause C, because it may have been instantiated as a result of solving some of the goals in the clause body. In order to preserve it as it is, a special copy of this clause instance is obtained. This copy is preserved in the same form, irrespective of which goals are solved in the following.

The desired instance Cd is obtained from the faulty instance Cf. The current goal is matched against the corresponding goal in the given goal trace and so the faulty instance Cf will change into the desired instance Cd. Again, a copy of this instance is obtained to preserve it in the same form.

In the following section we shall show how the error information stored is used in error correction.

4.3 CORRECTION OF INSTANTIATION ERRORS

We have mentioned before that no errors are corrected while search for a solution still continues. This is because we want to be sure that the right branch in the search tree has been followed before any modifications are made. But as soon as the search has terminated the error correction is initiated. The error information stored is used in the process.

Each particular instantiation error is corrected by modification of one clause only. This is the 'faulty clause' C. The clause C is modified on the basis of Cf and Cd, the faulty and the desired clause instance. They are decomposed into subterms and while this is being done the system tries to find various 'variable instantiating predicates'. First, let us see how decomposition is performed.

Decomposition of Clause Instances

The purpose of decomposition is to identify variables in the faulty clause instance Cf. If a variable has been identified the system checks whether this variable needs to be instantiated and how this should be achieved.

The decomposition of the clause instances Cf and Cd and the 'faulty clause' C proceeds in parallel. So, if a variable is identified in the faulty instance Cf, the corresponding subterms are immediately available.

Decomposition of each particular clause (or clause instance) is quite straightforward. Each clause or clause instance is treated as a 'term'. Implication and conjunction operators are treated as ordinary operators (eg. +). The decomposition of clause 'P1 <- P2', for example, produces

'P1' and 'P2' as subterms. Similarly, the decomposition of 'P3 & P4' produces 'P3' and 'P4' as subterms. This process has already discussed (in chapter 3.5).

The subterms obtained by decomposition are used in the search for new 'variable instantiating predicates'.

Search for Variable Instantiating Predicates

Let us see how new variable instantiating predicates are obtained.
Suppose that the following subterms have been obtained by decomposition:

Xi - a subterm of the clause C,
Yi - a subterm of Cf,
Ti - a subterm of Cd.

If Yi is a variable the search for new variable instantiating predicates is initiated. The system looks for predicate 'P(Yi)', which can be solved as a 'goal' and which changes into 'P(Ti)' as a result. If such a predicate is found the predicate 'P(Xi)' is used as the desired variable instantiating predicate.

The predicates which can be used as 'variable instantiating predicates' are given to the system initially. They may be either unary predicates, that is predicates with one argument, or binary predicates, that is predicates with two arguments.

The binary predicates are dealt with in a similar way. Each triple of subterms obtained by decomposition is stored so that it can be used with each new triple of subterms encountered later.

Suppose that the following subterms are just being considered:

Xi,Xj - subterms of clause C,
Yi,Yj - subterms of Cf,
Ti,Tj - subterms of Cd.

The system looks for a predicate 'P(Yi,Yj)' which when regarded as a goal and solved changes into 'P(Ti,Tj)'. If such a predicate is found the predicate 'P(Xi,Xj)' is used as the variable instantiating predicate.

If several variable instantiating predicates have been found this way, a disjunctive expression is generated containing all the alternatives as disjuncts. The disjunctive expression is used to modify the 'faulty clause'.

Modification of the Faulty Clause

The faulty clause is modified in the following way. The new variable instantiating predicates generated are added to the left of the symbol '!'. This has the following effects: The variable instantiating predicates are used in the clause selection phase in a similar way as 'constraints' even though they do not actually constrain selection. Also, they are not used in the process of detection of errors. That is they are ignored when the goals obtained by the system are compared with the goals in the given goal trace.

Example

In this example we shall illustrate how new variable instantiating predicates are generated. Suppose that the faulty clause C and the clause instances Cf and Cd have been obtained by the system.

Suppose they are as follows:

```
Faulty clause    C :     series(X1:X2)   <-  ! & write(X3)
Clause instance  Cf :    series((a:b):c) <-  ! & write(X3)
Clause instance  Cd :    series((a:b):c) <-  ! & write(d)
```

Suppose that predicate 'next(X1,X2)' is the only predicate which can be used as a variable instantiating predicate here. This predicate is 'true' if 'X2' is the next letter after 'X1'.

To find the variable instantiating predicates the clause C and the instances Cf and Cd are decomposed into subterms. After several steps the following subterms are obtained:

X3 - subterm of Cf
d - subterm of Cd

As the subterm of Cf is a variable, the search for new variable instantiating predicates is initiated. The subterms shown are used with various subterms stored previously. The following subterms are considered among others:

X3,c - subterms of Cf
d,c - subterms of Cd

Next, the predicate 'next' is used in conjunction with these subterms. The system tests, for example, if the predicate 'next(c,X3) can be solved and if it can, whether it can change into 'next(c,d)'. As both conditions are satisfied the predicate 'next(X2,X3)' is used as one of the variable instantiating predicates.

The predicate 'next(X2,X3)' obtained in this way is added to the left of the symbol '!' in clause C. The following clause is obtained:

```
series(X1:X2)  <-  next(X2,X3) &  ! & write(X4)
```

More examples will be given in section 7.

-*-

Instantiation of Several Variables

If several variables need instantiating the variables are dealt with one by one. They are simply dealt with in the order in which they are encountered. The set of variable instantiating predicates found in each step is added as a conjunct to other predicates found before.

This simple method of dealing with several variables may need to be extended on the basis of further work. This is because the order in which the variables are considered may affect the final result. That is different variable instantiating predicates may be obtained in each case. But if there are different ways of instantiating the variables dealt with, then they should all be found by the system.

Example

Suppose that the faulty clause below is to be modified on the basis of the clause instances shown:

```
Faulty  clause   C :     series(X1:X2) <-  ! & write(X3:X4)
Faulty  instance Cf:     series(a:b)   <-  ! & write(X3:X4)
Desired instance Cd:     series(a:b)   <-  ! & write( d:c)
```

Suppose that the predicates 'next(...)' and '..=:..' are to be used as variable instantiating predicates here. Also, suppose that clause 'n2' shown has higher priority than clause 'n3'.

```
n2: next(b,c) <- !
n3: next(c,d) <- !
```

We see that two variables in Cf need instantiating (X3,X4). If the variables are dealt with in the order in which they are encountered the following predicates are generated:

```
d=:X3        &  (c=:X4 v next(X2,X4)).
```

If the variables were dealt with in the opposite order the following predicates would have been obtained:

(c=:X4 v next(X2,X4)) & (d=:X3 v next(X4,X3)

Both ways of instantiating these two variables should be found by the system.

-*-

4.4 ELIMINATION OF ADDITIONAL SELECTION ERRORS

In this section we shall describe how both selection and instantiation errors are dealt with if they are detected with one goal. First let us consider why both types of errors can occur with one and the same goal.

Suppose a conflicting selection error has been detected. Detection of a conflicting selection error is always accompanied by reselection. The clause (or clauses) that may be selected may not be quite right and an instantiation error may be detected as a result. Both errors are associated with one and the same goal.

Both errors are corrected by modification of one and the same clause. In our system the instantiation error is dealt with first. This error is corrected by addition of a new variable instantiating predicate to the 'faulty clause' in the way described in the previous section.

The selection error is corrected by conflict resolution. The 'faulty clause' dealt with before is modified again. New constraints are generated and added to the variable instantiating predicates generated before. In our system any new predicates generated are always added to the left of the

existing predicates. This could be extended.

The system could be extended so that it would insert new predicates where appropriate. Perhaps explicit orderings for predicates could be used in a similar way as we use priority orderings for clauses. The advantages and disadvantages of this are yet to be investigated.

4.5 SOME PROBLEMS WITH VARIABLES

Certain type of instantiation errors cannot be detected by the system as it is currently implemented. The inability of our system to detect these errors is related to the way variables are represented and handled during the search for the solution.

In this section we shall describe how the difficulties can be overcome. A technique of 'freezing' instantiations in the term specified will be described.

Let us first see what type of instantiation errors the system cannot currently detect. Problems of detection arise if several variables are used in the goals obtained by the system, and if the goals belong to different 'nodes' in the search tree. The system cannot determine if one and the same variable is correctly used in these goals.

Example

Let us consider the goals in the following figure. The goals in the given goal trace contain two occurrences of one variable (X1). The goals obtained by the system, however, contain two different variables

(X1,X2). The system as it is implemented cannot detect this.

Step	Given Goal Trace	Goals Obtained by the System
1	X1 = 4-2	X1 = 4-2
2	4-2 = X1	4-2 = X2

Fig. 4.4 Problem of Detection of Errors

-*-

Let us see why this type of error is difficult to detect. It is related to the way variables are handled by the system during search.

After each clause has been applied, the existing variables are systematically replaced by 'new' variables which are not used elsewhere (see chapter 2.3.) This action helps to localize the effects of clause application to the one branch in the tree which is currently being followed. As goals on one branch in the tree are being instantiated, the goals on other branches do not change.

We chose this particular way of implementing search since it was particularly easy to implement. We used the meta-predicate 'assert' in Prolog (Warren, 1977) to store the goals obtained by application of each clause. All variables were automatically replaced by new ones.

Unfortunately, this technique prevents the detection of the particular type of instantiation errors mentioned. As new variables are introduced in each step, the system cannot determine whether the same variable has been used in different nodes in the search tree. In the following we shall describe how the difficulties can be overcome.

How Can the Difficulties be Overcome ?

Let us see how variables should have been handled during the search for the solution. What we want is that all the variables belonging to each new node generated are left intact until the node is dealt with again. No substitutions for the existing variables should be performed until this is explicitly permitted. Let us see how we could implement this.

Implementation in Prolog is not easy. This is because in Prolog it is impossible to specify that the variables in a particular term should be left as they are, irrespective of whether terms are being substituted for these variables elsewhere. Also, the existing primitives in Prolog do not allow this to be specified very easily.

One possibility we have is to use a different representation for the variables which used in the algebraic expressions dealt with. Currently these variables are represented by Prolog variables, but they could be represented by constants.

For example, instead of using 'X1=4-2', we could use 'x1=4-2'. The only difference between the two goals is that the Prolog variable 'X1' is used in one goal, while the constant 'x1' is used in the other goal.

This representation allows us to implement the search in the way we want. First, we would develop a facility of 'freezing' a term and 'retrieving' it, which we will now describe.

Each 'frozen' term would be left intact. The variables in it would be left unmodified, even if the same variables were instantiated elsewhere.

That is a copy of the 'variable substitution list' (*) would be obtained after the term has been 'frozen'. This copy would not be updated irrespective of whether the original is updated or not, while the term is 'frozen'.

Each 'frozen' term could be 'retrieved' in the following way. The first 'frozen' term matching the term specified would be retrieved. The instantiations performed elsewhere while this term was 'frozen' would be ignored. The facility of 'freezing' and 'retrieving' terms would be used in the implementation of search.

We believe that 'freezing' a term and 'retrieval' of such terms are two basic functions which would be generally useful, and that they should be provided in a future version of Prolog. We believe that it would substantially strengthen its power.

4.6 IMPLEMENTATION

The implemented system is capable of detecting and correcting instantiation errors in the way described. We have mentioned before that each instantiation errors is corrected on the basis of the faulty clause instances Cf and the desired clause instance Cd.

Currently, the clause instances Cd and Cf have to be given to the system manually. Relatively small extensions are needed so that the system could determine what these clause instances look like without outside intervention.

(*) The 'substitution list' is a list of variable-term pairs determining how the term is to be instantiated.

We have already explained how the clause instances can be obtained. We have mentioned that what the system has to do is mark each goal after it has been solved but not delete it from the goal stack. The goals stored enable the system to reconstruct the clause instances needed without any difficulties.

Suppose that 'series((a:b):c)' is the current goal and suppose that the system has chosen to use the clause

C: series(X1) <- ! & write(X2).

Suppose that subsequently an instantiation error was detected. At this moment the goal stack will contain the following goals:

write (X2)
series((a:b):c) (solved)

The faulty clause instance

Cf: series((a:b):c) <- ! & write(X2)

can be easily obtained from these two goals.

The desired clause instance Cd is obtained from the faulty clause instance Cf by matching. As the current goal is matched against the corresponding goal in the given goal trace the clause instance Cf will change into Cd.

The examples dealt with by our system were relatively simple. This is why this simple method of obtaining the clause instances seems quite adequate. Storing each clause instance is not too economical, however. It would be better if we obtained only the list of substitutions from which each clause instance could be obtained. This would save storage. The clause instance Cf, for example, can be obtained from clause C, if 'X1' is replaced by '(a:b):c'.

4.7 EXPERIMENTAL RESULTS

Two sets of problems were studied in detail. Both dealt with letter series completion. We studied how our system would acquire the ability to instantiate variables.

The problem of letter series completion has been studied by others. Most of the programs were special purpose programs (Simon and Kotovsky, 1961), or they relied on special purpose heuristics (Waterman, 1970). None of these programs could be easily extended so that it would deal with simple equations. Waterman's work is discussed in chapter 6.

4.7.1 Letter Series aabb.

The ability to predict the next letter in this series was acquired in several steps. In each step an example of a particular series was given, eg. 'aa', 'aab' etc., and the problem was to determine what the next letter is. If the letter was not correct, modifications were performed to the existing clauses. After several steps the system was able to predict the next letter correctly.

Information Given to the System

Each particular letter series is represented by a term. The series ' aabb', for example, is represented by the term '(((-:a):a):b):b'. The symbol '-' represents a blank (a blank space). Each term representing a series can be used as an argument of the predicate 'series'. This way we can obtain, for example, the following predicate:

```
series((((-:a):a):b):b)
```

Each predicate 'series(..)' when given to the system a goal initiates the search for a solution. What exactly happens depends on which clauses are available to the system at the time. In this series of experiments only one clause was given to the system initially. It was the following clause:

```
s0:  series(X1) <-  !  &  write(X2).
```

This clause expresses the following:

If 'X1' is the given series, then 'X2' should be 'written out' as the next letter. That is, the series should continue with the letter 'X2'.

Several new versions of this clause were generated by the system later. The modifications were of two types. Selection errors were corrected by addition of constraints. Instantiation errors were corrected by addition of new variable instantiating predicates. The following predicates were used by the system in both stages:

- next(Xi,Xj),
- Xi =: Xj.

Predicate 'next(Xi,Xj)' is true, if 'Xj' is the next letter after 'Xi'. Predicate 'Xi =: Xj' is true if 'Xi' and 'Xj' are identical, or if they can be made identical by instantiating the variables in 'Xj'. This predicate has been used in other examples shown before.

A number of clauses were also given to the system determining which letter follows which. For example:

```
next(a,b)   <-  !,
next(b,c)   <-  !  etc.
```

Goals Given to the System

The goals given to the system are shown in Fig. 4.5. Each goal is a particular example of the series 'aabb'. Each goal when given to the system initiated the search for a solution. The aim was to find the next letter in the given series.

Step :	Given Goal :	
1	series (-:a)	) Errors occurred
2	series ((-:a):a)	) in these steps
3	series (((-:a):a):b)	) The next letter was
4	series ((((-:a):a):b):b)	) correctly predicted

Fig. 4.5 Goals Given to the System

Errors were detected by the system with goal no. 1 and 2. After all the errors have been corrected the system was given goal no. 3 and 4. Due to the modifications performed before no errors occurred with the last two goals. Each step will now be described in detail.

Step 1

The first example of the series 'aabb' given to the system was represented by 'series(-:a)'. Clause 's0' was the only clause available at this stage, and this clause was also selected and applied. The goal obtained by the system was compared with the corresponding goal in the given goal trace, and an instantiation error (IE) was detected as a result (see Fig. 4.6). As no new subgoals were generated later, when 'write(X1)' was dealt with, the search was terminated in the next step.

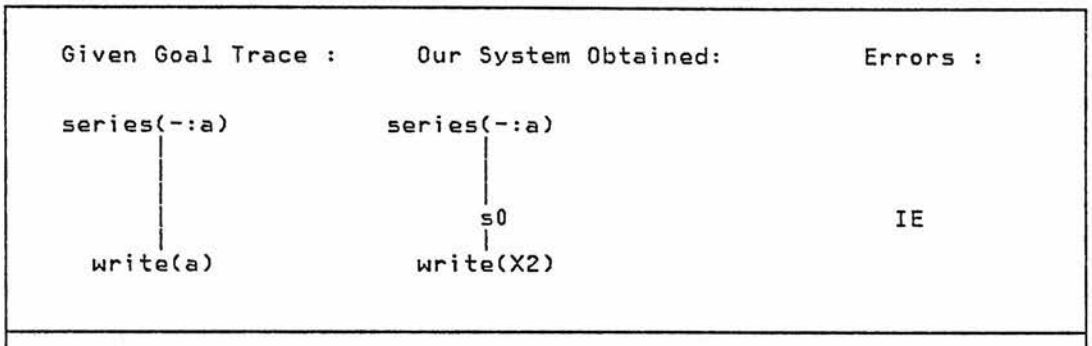


Fig. 4.6 Search for a Solution

Let us see how the instantiation error was corrected. Clause 's0' was identified as the 'faulty clause' which must be modified. The following two instances of that clause were analysed by the system:

Desired Instance: series(-:a) <- ! & write(a),
Faulty Instance: series(-:a) <- ! & write(X2).

The aim was to identify the variables in the faulty clause instance which should have been instantiated. We see that variable 'X2' should have been instantiated to 'a'.

In the next step the system tried to find the predicates which would instantiate 'X2' in the desired way. Two different ways of instantiating this variable were found by the system, and this is why a disjunction of predicates was added to clause 's0'. The new clause obtained was as follows:

```
s1:  series(X1) <- (a=:X2  v X1=:(..:X2))  & !  & write(X2)
```

The new clause 's1' contains two new disjuncts. One instantiates 'X2' to 'a'. The second one uses the last letter in the given series as the letter to be written out (X2).

Step 2

The next goal given to the system is shown in Fig. 4.7. First, the system selected clause 's1' in the attempt to solve it. This clause was also applied and so goal 'write(a)' was obtained. This goal, however, does not match the corresponding goal in the given goal trace, and this gave rise to a conflicting selection error (CSE). Next, clause 's0' was selected and then an instantiation error was detected.

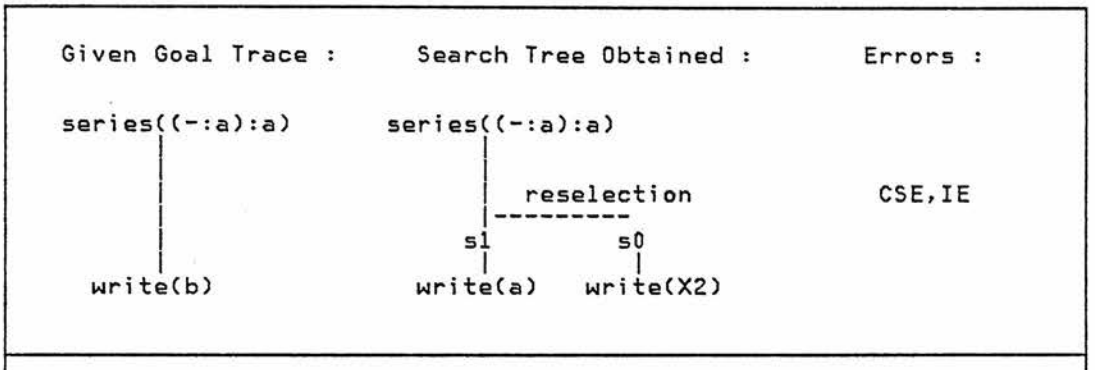


Fig. 4.7 Search for a Solution

The instantiation error detected was dealt with first. The following clause instances were used in the process:

```
Desired Instance:    series((-:a):a) <- ! & write(b),
Faulty  Instance:    series((-:a):a) <- ! & write(X2).
```

The new clause version 's2' was generated in a similar way as clause 's1' shown before. The new clause was as follows:

```
s2: series(X1) <-  ( b =:X2                                  v
                     X1=:     (..:L1) & next(L1,X2)          v
                     X1=:((..:L2):..) & next(L2,X2)          )
                     &  !             & write(X2).
```

Clause 's2' contains three new disjuncts. The disjunct 'X1=:(..:L1) & next(L1,X2)' (*), for example, uses the last letter in the given series

(L1) to derive the value of the next letter (X2). The other two disjuncts represent two other hypothesis about how the next letter in the series should be obtained.

After all this was done, the system dealt with the conflicting selection error (CSE). The clause 's2' generated in the previous step was modified again. The following error information is used in the process:

Selection context: series(-:a:a),
Rejection context: series(-:a).

All the predicates added to clause 's2' before were regarded as 'constraints' by the system, and an attempt was made to modify them. Two disjuncts which were found 'true' in the rejection context were deleted. The method of constraining selection this way was discussed in chapter 3.7. The modified version of 's2' was as follows:

```
s2: series(X1) <-  X1=: ((..:L2):..)   &  next(L2,X2)
                   &  !                &  write(X2)
```

This clause was given priority over 's1'. The system of priority orderings obtained was as follows:

s2 > s1 > s0

(*) Each variable Li used here was chosen to represent a particular letter in the given series; L1 is supposed to represent the last letter, L2 is the letter just before L1 etc.

Step 3 and 4

The goal given in this step is shown in Fig. 4.8. Clause 's2' was correctly selected in this step and no new modifications were needed.

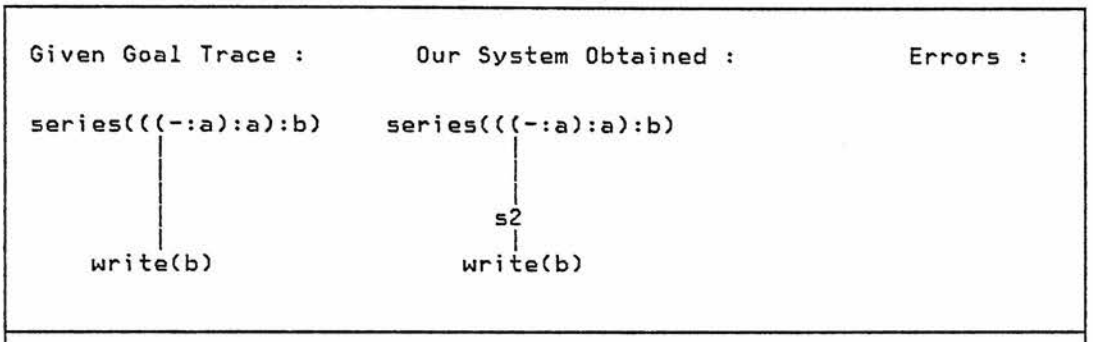


Fig. 4.8 Search for a Solution

The goal in Step 4 was also solved without errors using clause 's2'.

What the System Has Learned

Clause 's2' which gave us correct results with goal no. 3 and 4 expresses what the system has learned. The clause tells us that we should take the given series (X1) and use the second letter before the end (L2) to derive the value of the next letter (X2). That is, 'X2' should be the successor of 'L2' (the next letter in the alphabet after L2).

4.7.2 Letter Series 'abmcdm'.

This letter series was dealt with in a similar way as the previous series. The same clause was given to the system initially:

```
z0: series(X1) <- ! & write(X2).
```

The same type of predicates were used to modify the clauses dealt with. The following figure shows the goals given to the system. Each goal shows an example of the particular series used in this example.

```
No.:      Given Goal :

 1        series (-:a)                          )
 2        series ((-:a):b)                      )
 3        series (((-:a):b):m)                  )
                                                )  Errors occurred
 4        series ((((-:a):b):m):c)              )  in these steps
 5        series (((((...):b):m):c):d)          )
 6        series ((((((...):m):c):d):m)         )

 7        series ((((((...):c):d):m):e)         )  The next letter was
 8        series ((((((...):d):m):e):f)         )  correctly predicted
 9        series ((((((...):m):e):f):m)         )
```

Fig. 4.9 Goals Given to the System

Errors occurred with the goals 1 - 6, but correct results were obtained afterwards. All clauses were generated by the system in a similar way as in the previous example. All the clauses obtained are shown in the following figure.

```
z1:  series(X1) <-  ( b=:X2        v X1=:(..:L1)  & next(L1,X2) )
                      &  !                        & write (X2)

z2:  series(X1) <-  ( X1=:(..:b)  v X1=:((..:..):..)              v
                      X1=:((..:L2):L1)            & next(L2,L1))  &
                      m=:X2  &  !                 & write (X2)

z3:  series(X1) <-  ( X1=:(..:m)                  v  X1((..b):..) v
                      X1=:(((..:..):..):..)                       v
                      X1=:(((..:L3):L2):..)       & next(L3,L2) ) &
                    ( c=:X2  v  X1=:((..:L2):..) & next(L2,X2) )
                      &  !   &  write(X2)

z4:  series(X1) <-  ( X1=:(..:c)                  v ... v
                      X1=:     (((..:L3):..):L1) & next(L3,L1) v
                      X1=:((((..:L4):L3):..):..) & next(L4,L3) ) &
                      X1=:                (..:L1) & next(L1,X2)
                      &  !                        & write (X2)

z5:  series(X1) <-  ( X1=:       ((..:L2):L1)     & next(L2,L1) &
                      m=:X2  &  !                 & write(X2)

z6:  series(X1) <-  ( X1=:(..:m)  v
                      X1=:(((..:L3):L2):..)       & next(L3,L2) ) &
                      X1=:     ((..:L2):..)       & next(L2,X2)
                      &  !                        & write (X2)
```

Fig. 4.10 Clauses Obtained by the System

The priority orderings that were obtained by the system were as follows:

```
z5 > z4 > z3 > z2 > z1 > z0
z6 > z4
```

Fig. 4.11 Priority Orderings Obtained

What the System Has Learned

The system has acquired six new clauses in this example. In the first three steps the system acquired clauses z1-z3. These clauses were subsequently modified and the new versions are called z4-z6 (z4 is a new version of z1 etc.). The three later versions will be used by the system if we give it another example of this series. Let us now see how the information acquired by the system can be interpreted.

Use the letter before the end

The second letter before the end (L2) may be used to derive the next letter in the series (X2). The next letter is the successor of 'L2'.

This rule may be invoked either if the last letter is 'm', or if the second letter before the end (L2) is the successor of the letter preceding it (L3).

Clause 'z6' obtained by the system expresses just this.

```
z6:  series(X1) <-  ( X1=:(..:m)  v
                      X1=:(((..:L3):L2):..) & next(L3,L2) ) &
                      X1=:     ((..:L2):..) & next(L2,X2)
                      &  !                  & write(X2)
```

Clause 'z6' is capable of extending, for instance, the series 'mefm'

correctly. The next letter obtained is 'g'.

Sometimes the next letter is 'm'

The next letter in the series will be 'm', if the last letter in the series (L1) is the successor of the preceding letter (L2). This is expressed by clause 'z5' which has been acquired by the system.

```
z5:  series(X1) <-  ( X1=:((..:L2):L1) & next(L2,L1) &
                      m=:X2  &  !       & write(X2)
```

This clause extends series 'dmef', for example, correctly. The next letter is 'm'.

Sometimes use the last letter

The last letter in the series (L1) may be used to derive the next letter (X2). The next letter should be the successor of that letter (letter L1). However, one of several other conditions must be satisfied. For example, the last letter in the series (L1) must be the successor of one of the preceding letters - the third letter before the end (L3). Other conditions are similar. This is expressed by clause 'z4' obtained by the system.

```
z4:  series(X1) <-  ( X1=:(((..:L3):..):L1) & next(L3,L1) v ...) &
                      X1=:          (..:L1) & next(L1,X2)       )
                      &  !                  & write(X2)
```

The rule mentioned should be used only if the two preceding rules cannot be used. That is, the priority orderings 'z5 > z4' and 'z6 > z4' should be respected.

This clause correctly extends the series 'cdme', for example. The next letter predicted is 'f'.

4.7.3 Example from the Arithmetic Domain

The examples discussed earlier were from the domain of letter series completion. The techniques of detection and correction of instantiation errors described in this chapter can be applied to the domain of arithmetic and algebra. The next example shows how a typical error can be corrected.

Let us suppose that an instantiation error has occurred. Suppose that the 'faulty clause' below has been obtained by the system together with the clause instances shown:

Faulty clause:	X1 = X2	<-	!	&	X3 = X4
Faulty clause instance:	3 = 5-2	<-	!	&	X3 = X4
Desired clause instance:	3 = 5-2	<-	!	&	5-2 = 3

In this example two variables in the 'faulty clause instance' need instantiating (X3,X4). Instantiation of each variable is ensured by the addition of new variable instantiating predicates to the existing clause. The following clause is obtained:

```
New clause:   X1 = X2   <-  ((5-2=:X3  v   X2=:X3)  &
                             (3=:X4    v   X1=:X4))
                             &   !     &   X3 = X4
```

Correction of errors like this is quite straightforward. Correction of instantiation errors in the algebraic domain is more difficult because of the problem with variables, discussed in section 5.

4.8 SUMMARY

In this chapter we have discussed how instantiation errors are detected and corrected. We have described what type of error information is stored when the errors are detected and how this information is used later, when the errors are corrected.

The error information used includes the name of the 'faulty clause', and two particular instances of that clause. In section 3 we have shown how the clause instances are examined and how the appropriate variable instantiating predicates are found. The new predicates are then added to the existing predicates in the 'faulty clause'.

If both selection and instantiation errors are detected with some particular goal, they are both corrected by the modification of one clause. The new predicates, that is new predicate constraints and new variable instantiating predicates are added to the left of the existing predicates. It seems desirable to allow the predicates to be inserted where appropriate.

We Deal with one Clause Only

Our system is based on an assumption that each instantiation error could be corrected by modification of just one clause. We have also assumed that it is sufficient to examine the instances of the clause to be modified in the process. Further extensions might need to be made in this area on the basis of further work.

Problems with Variables

We have also pointed out that certain types of instantiation errors cannot be detected by the system, as it is implemented. We have explained that it cannot detect if one and the same variable is used in different nodes in the search tree. We have pointed out that the problem arises because of the way search has been implemented in in our system.

We have described how this particular problem could be overcome. We have described a technique of 'freezing' the instantiations in the term specified, which is useful for this purpose. Because of its general usefulness, we have suggested that this facility should be provided in future versions of Prolog.

Experimental Results

Two sets of problems were studied in detail. The problems were taken from the domain of letter series completion. Each set of problems dealt with a particular letter series. Five instantiation errors and four conflicting selection errors were detected with these problems. They were all corrected in the way described in Chapters 3 and 4. After all these errors were corrected, other examples of the particular series chosen could be completed without errors.

A short example is given, illustrating that instantiation errors in the domain of arithmetic and algebra could be handled in a similar way.

5 EXTENDED SYSTEM ELM2

5.1 INTRODUCTION

In this chapter we shall describe the extended system ELM2 in which it is possible to control selection of clauses more effectively. Various goals awaiting solution are taken into account before deciding what is to be done in each step.

Several other researchers have been concerned with the problem of how a set of goals should be solved. Sussman (1975), for example, has shown that certain problems can be solved only if the given goals are solved in a certain order. His planning system can find what the correct order is.

Kowalski (1979) has been concerned with how a set of goals should be solved. He has pointed out that different strategies for deciding which goal is to be solved next often affect the efficiency of execution.

Burstall and Darlington (1975) have described how the given program can be transformed so that two or more goals could be solved in an efficient way. They have pointed out that first we should examine how each goal is solved. This enables us to identify various common steps which can be eliminated later.

Criticism of ELM1

In our system ELM1 which we have described before only one goal is taken into account at any given point in time. This is the topmost goal on the stack. This goal determines which clause should be selected next. All the other goals on the stack are ignored until the first goal has been solved.

With many problems it is quite sufficient to do this. Several such problems were shown before. All of the problems shown could be solved without errors in the end, that is after all the errors detected have been corrected.

Sometimes, however, all of the goals that need to be solved matter. The action to be taken depends just on what these goals are. Let us consider, for example, the following two problems:

Problem 1:	Problem 2:
(X-1)*(X-1) =Y & Y+4*X = 4	(X-1)*(X-1) =Y & sqrt(Y)=3

Equations can often be solved in several different ways. Suppose that this is how we want the equations to be solved:

Solution of Problem 1:	Solution of Problem 2:
(X-1)*(X-1) =Y & Y+4*X = 4	(X-1)*(X-1) = Y & sqrt(Y)=3
X**2 -2*X +1 =Y & Y+4*X = 4	(X-1)**2 = Y & sqrt(Y)=3
X**2 -2*X +1 + 4*X = 4	sqrt((X-1)**2) =3
X**2 +2*X -3 = 0	X-1 =3 or X-1 =-3
X = 1 or X =-3	X = 4 or X =-2

We see that a different action needs to be taken in the first

step even though the first goal in each conjunction is the same. A particular method of division which is described in detail in section 6 could not be learnt by the system if various goals awaiting solution were not taken into account in the process.

The problem of dividing 'X1' by 'X2' is solved by splitting up 'X1' (the dividend) into a number of integers which are equal to 'X2' (the divisor), and by counting how many integers have been obtained in this way. Suppose that our problem was, for example, 4/2=X1. We see that '4' can be split up into two 2's, and so the answer is two. Figure 5.5 on page 185 shows how the problem is dealt with step by step. First, the given problem 4/2=X1 is transformed into

$$0+4 = X2 \quad \& \quad X2/2 = X1,$$

and then '1' is repeatedly subtracted from the second integer (4) and added to the first integer (0). This process of adding and subtracting '1' is repeated until the first integer is big enough to match the divisor:

```
2+2 = X2   &   X2/2 = X1.
|                   |
----- match -------
```

We see that the process of adding and subtracting '1' could not be terminated correctly if both goals were not taken into account.

In our extended system, referred to as ELM2, various goals awaiting solution affect selection of clauses and this is why ELM2 is more powerful than ELM1.

The clauses used in ELM2 are different from the clauses used in ELM1, because several predicates may appear in the head of each clause. The clause can be selected only if the clause head can match the goals awaiting solution (the current goal stack). The new type of clauses will be referred to as goal stack clauses (GS clauses) in the following.

We shall show later how these clauses can be generated by the system. We shall see that the contexts used in the process must include the whole goal stack.

5.2 GOAL STACK (GS) CLAUSES

In this section we shall describe the new type of clauses that we use in our extended system (ELM2). Also, we shall describe how these clauses are used during selection.

The clauses which we have used in ELM1 were written in the following form:

G <- Cs & ! & Rs, where

G represents the clause head,
Cs represents constraints and
Rs represents predicates in the clause body.

Only one predicate could appear in the clause head of each clause. The new type of clauses which we use in ELM2 appear in the following form:

```
G & Gs <- Cs & ! & Rs.
```

As in ELM1, the symbol 'G' represents a predicate. The symbol 'Gs' represents either a predicate or a conjunction of predicates. Any conjunct in that conjunction may be left uninstantiated (the variable 'Gs' may also be left uninstantiated).

Let us now see how selection is performed in ELM2. Each GS

clause can be selected only if the clause head can match the current goal stack. That is the predicate 'G' must match the topmost goal on the stack and 'Gs' must match the remaining goals. The goals are used in the order in which they appear, that is they cannot be reordered during selection process. If the match has succeeded the predicate constraints and priority orderings are also taken into account, as described in chapter 2.2. The predicates constraints may refer to any subterms in the clause head.

Selection of the new type of clauses is affected by the goals that appear on the goal stack. This is why the new type of clauses used here are referred to as 'goal stack clauses' (GS clauses).

As each GS clause is applied, only one goal is, in effect, solved. This is the topmost goal on the goal stack. This goal is ignored in the future search.

The main advantage of using GS clauses is that they allow the search to be controlled more effectively. Selection of such clauses is affected not only by which predicates are used in the clause head, but also by their constraints. In the following section we shall show how the GS clauses are constrained by our system.

5.3 GENERATION OF GS CONSTRAINTS

In our system new clauses are generated on the basis of 'contexts' stored automatically by the system. In our system ELM1 which was described before each context stored consisted of just one goal. The contexts stored by ELM2, however, include the whole goal stack. These contexts are used in the

process of generating new constraints later.

Apart from that the contexts may include more than one goal the constraints of GS clauses are generated in the same way as already described. Analysis of contexts is followed by the search for new constraints etc.

Example

Suppose that a conflicting selection error has been detected and that clause 'sqre' shown below is to be constrained.

```
sqre:  X1*X1=X2  &  Gs   <-  !  &  X1**2=X2  &  Gs
```

Suppose that the only predicate which can be used as a constraint here is ' =: ' described on page 108. Suppose that the following contexts have been provided by the system:

```
Selection context:   (X-1)*(X-1)=Y   & sqrt(Y)=3
Rejection context:   (X-1)*(X-1)=Y   & Y +4*X =4
```

The constraints are generated by conflict resolution. The new constraints generated are as follows:

```
Gs =: sqrt(..)=..   v  Gs =: sqrt(X2)=..   v  Gs =: ..=3
```

All the constraints generated refer to the second goal on the goal stack. The first constraint specifies that this goal must be an equation with 'sqrt(..)' on the left hand side. The second constraint is similar. The third constraint specifies that the equation must have '3' on the right hand side. The new version of clause 'sqre' generated contains the constraints shown. We notice that this clause can be selected in the first step during the solution of Problem 2 shown before

```
(X-1)*(X-1) = Y  & sqrt(Y)=3
```

but it cannot be selected with Problem 1:

```
(X-1)*(X-1) = Y  & Y+2*X =4
```

5.4 REORDERING GOALS WITH GS CLAUSES

In this section we shall discuss the use of GS clauses for reordering of the existing subgoals. The discussion presented here should provide a basis for further extensions of our system. None of the techniques described in this section were used in our experimental work which is described in section 6.

The ability to reorder the existing subgoals seems useful, because the difficulty of solving two or more subgoals is often dependent on which subgoal is solved first.

Example

Let us consider the problem of solving several simultaneous equations, such as the following:

```
(X-1)*2 + (Y-1)*2 = 3   &  X = 3
```

The solution of both equations is easier if the second equation is solved first.

Let us now see how we can extend our system so that it would be able to reorder the existing subgoals. What we have to do is extend the way the goal stack is modified after each clause has been applied. Each clause should specify what the new goal stack should look like. For example, the clause

```
G & Gs <-  Cs  & !  & Rs & Gs
```

shows that goal G should be removed from the goal stack and replaced by 'Rs' representing the new subgoals. Goals 'Gs' should be left intact. The following clause, for example, can interchange the subgoals G1 and G2:

```
G1 & G2 & Gs <-   Cs  & !  & G2 & G1 & Gs
```

The constraints of this clause (Cs) must be carefully chosen so that the two goals would not be repeatedly interchanged, however.

5.5 IMPLEMENTATION

The system ELM2 described in this chapter has not been implemented. However, very few changes are needed to transform the system ELM1 which has been implemented into ELM2.

First, we have to ensure that each 'context' stored by the system during the search for the solution includes the whole goal stack and not just one goal. This is easy, because the system knows what the goal stack looks like at any given point.

Next, we have to ensure that the goals stored are taken into account when new constraints are generated. The system ELM1 can already do that. Each context is treated as a term irrespective of whether it is an expression such as 'X1+X2', or a conjunction of goals 'G1&G2'. We see that the extensions are easy to make.

5.6 EXPERIMENTAL RESULTS

All the experimental results presented in this section were obtained by simulating ELM2 on paper. First, we were interested to see whether ELM2 would be able to learn to add integers, and then whether it would be able to learn division which is much more complex. We found that ELM2 could learn to do both tasks without difficulties.

Addition of Integers

We have decided to repeat our experiments with addition, described in chapter 3.14, to verify that the extended system ELM2 could learn to add integers equally well as ELM1. All clauses needed were, however, rewritten in the form of GS clauses. All the problems of addition which we have given to the system before were given to the system again. We were interested to see what would happen, and we found that all the errors detected were corrected in the same way as before.

Division of Integers

The next set of problems we have chosen dealt with division of integers. We have noticed that division could be performed in several different ways. We have studied one of these methods in detail and our aim will be to show how this method can be acquired by our system.

We know that the result of dividing X1 by X2 is some number showing how many times we can add X2 before we obtain X1. So, to perform X1/X2 we have to 'break up' X1 into a number of X2's and count how many times we did it. For example, if 4/2 is to be performed, the integer 4 has to be broken up into two 2's and consequently the answer will be two. Let us now see the algorithm in more detail.

The process of division can really be broken up into three phases as follows. In phase 1 The integer X1 is replaced by the sum of X2 and the rest (X4).

Phase 1		
Original expression:	X1/X2 = X3	(4/2 = X3)
New expression:	(X2+X4)/X2 = X3	((2+2)/2 = X3)

In phase 2 the expression '(X2+X4)/X2' is replaced by 'X2/X2 + X4/X2' which is then simplified into '1 + X4/X2.

Phase 2		
Original expression:	(X2+X4)/X2 = X3	((2+2)/2 = X3)
New expression:	1 + X4/X2 = X3	(1 + 2/2 = X3).

In phase 3 the subproblem X4/X2 is dealt with and the result (X5) is used in the original expression.

Phase 3		
Original expression:	1 + X4/X2 = X3	(1 + 2/2 = X3)
New expression:	1 + X5 = X3	(1 + 1 = X3)

Finally, the sum of 1 and X5 is calculated. More information about each phase will be given later. First, let us see which clauses we need in the process.

Clauses Used

Four new clauses seem to be needed to transform the goals in the way just described. They are shown in the following figure. All of these clauses were given to the system.

```
subsd:   X1/X2=X3        & Gs  <-   !  & X1=X4      & X4/X2=X3
izero:   X1=X2           & Gs  <-   !  & 0+X1=X2
distd:  (X1+X2)/X3=X4    & Gs  <-   !  & (X1/X3+X2/X3)=X4
cancd:   X1/X1=X2        & Gs  <-   !  & 1=X2
```

Fig. 5.2 New Clauses Needed

All the clauses obtained during the previous experiments with addition were also used here. These clauses are shown in the following figure.

```
subs1: X1+X2=X3        & Gs  <-   X1=:(..+..)  & ! & X1=X4 & X4+X2=X3
asoc:  X1+(X2+X3)=X4   & Gs  <-   !  & (X1+X2)+X3=X4

subs:  X1+X2=X3        & Gs  <-   !  & X1=X4    & X4+X2=X3
subz:  X1+X2=X3        & Gs  <-   !  & X2=X4    & X1+X4=X3
eq:    X1=X1           & Gs  <-   !

suc:   1+1 =X1         & Gs  <-   2=:X1    & !   etc.
pred:  2 =X1           & Gs  <-   1+1=:X1  & !   etc.
```

Fig. 5.3 Old Clauses Used

The new clauses were integrated with the old clauses using the method described in chapter 3.12. The new clauses were, in fact, given lower priority than some of the existing clauses. The additional priority orderings are shown in the following figure together with the orderings previously acquired.

suc > subs1 > asoc >	subz > subs	)	Orderings
	subz > eq	)	generated
	pred > eq	)	before
	subz > izero	)	Orderings
	pred > izero	)	added

Fig. 5.4 Priority Orderings Used for Division

Solution of 4/2=X1

The following figures show how the system used the given clauses in the search for the solution of 4/2=X1. Each figure depicts one of the three phases referred to before.

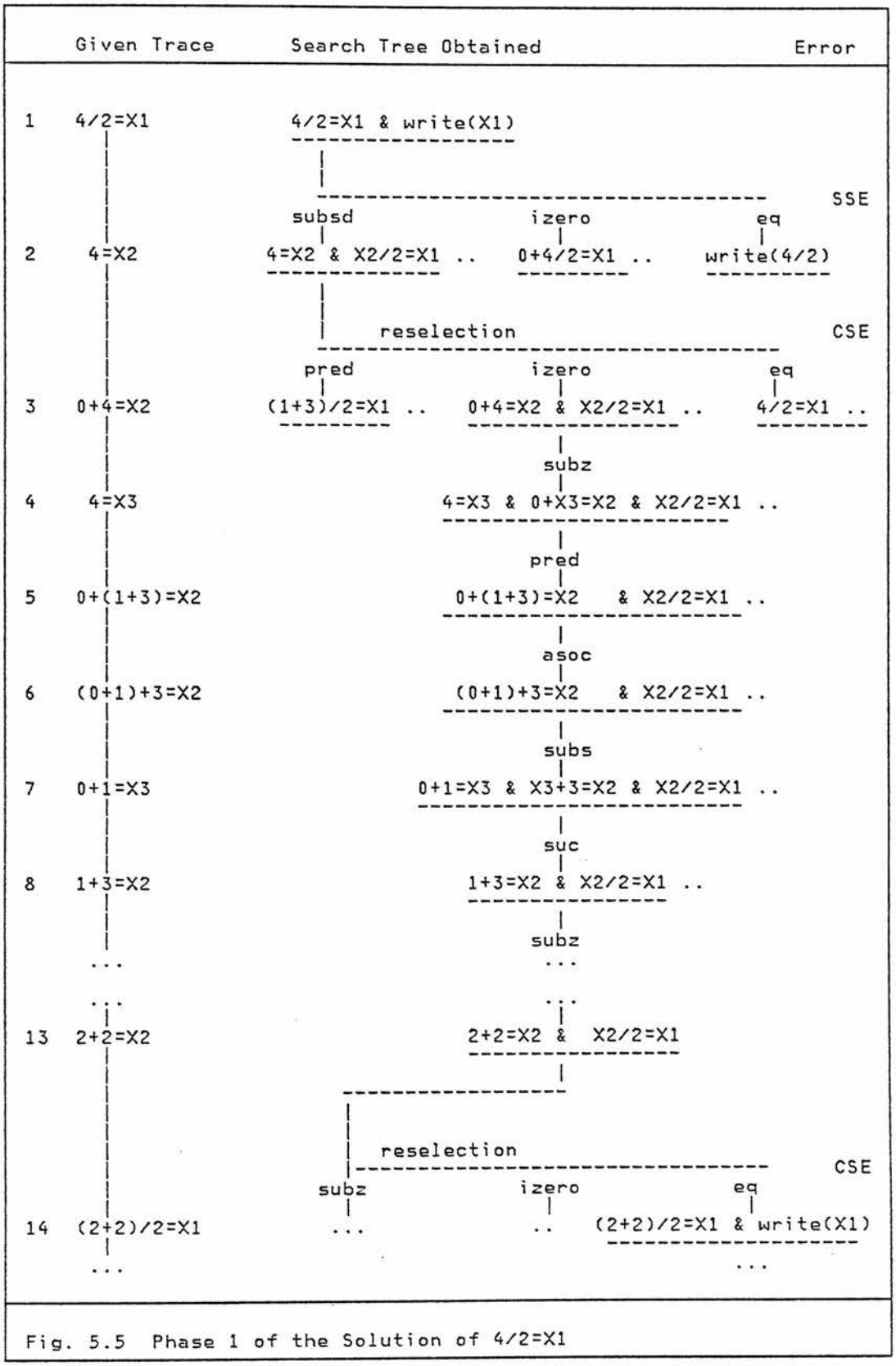


Fig. 5.5 Phase 1 of the Solution of 4/2=X1

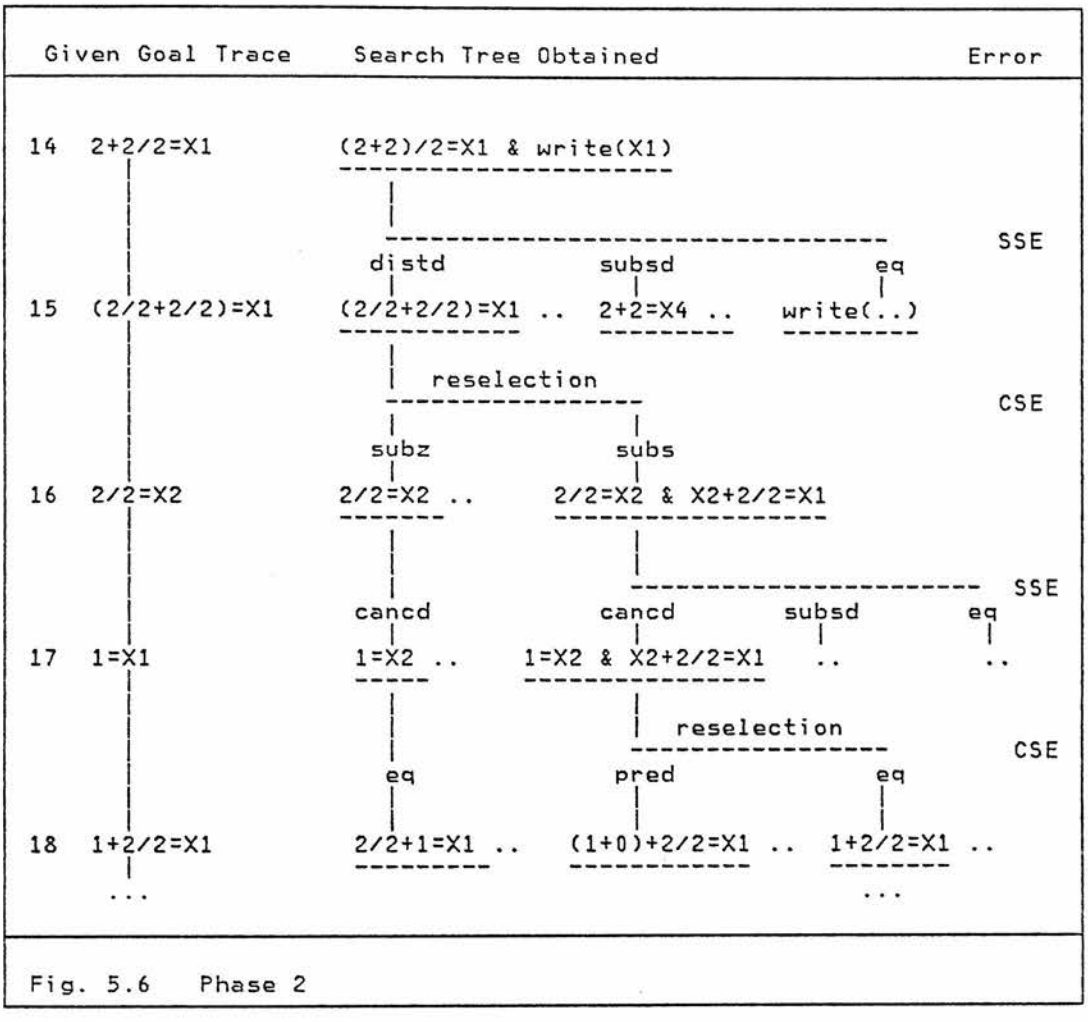


Fig. 5.6 Phase 2

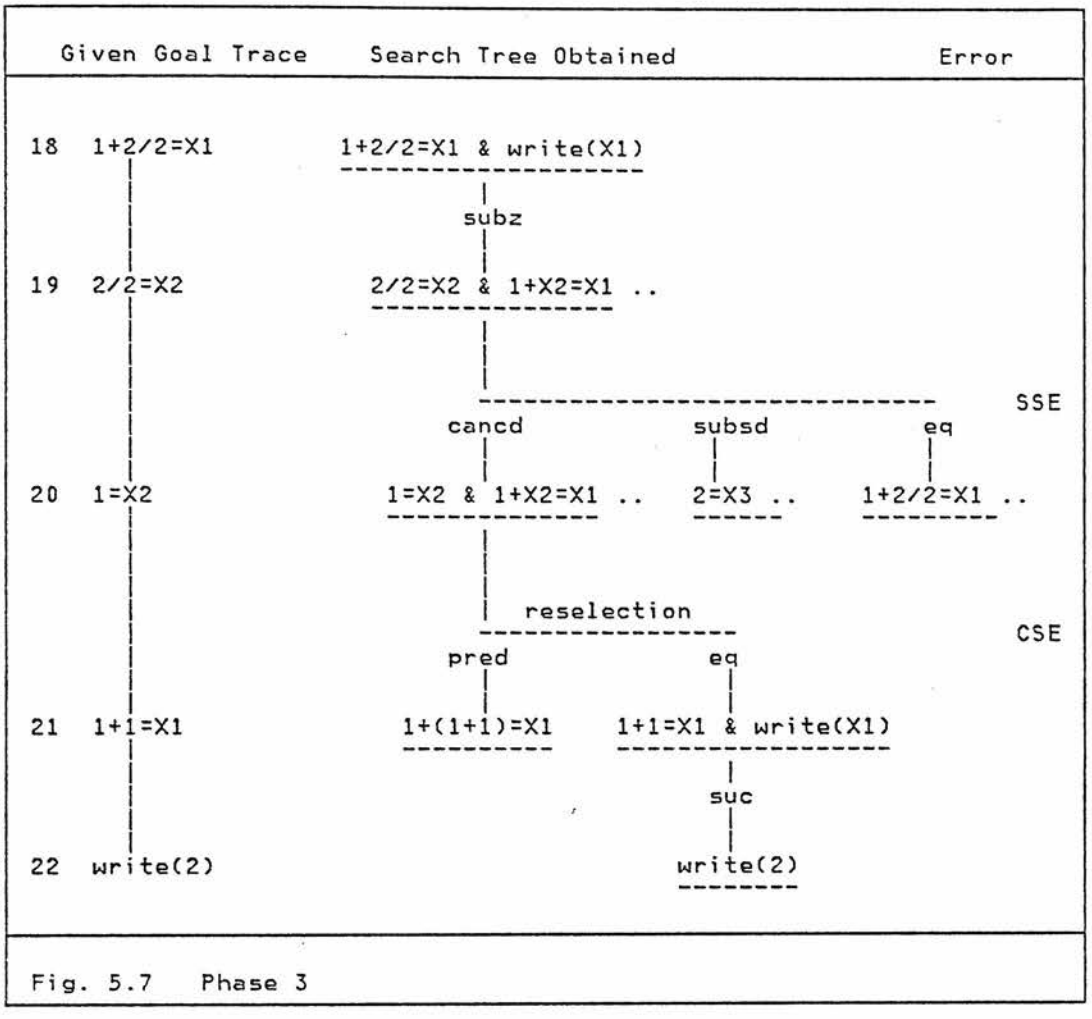


Fig. 5.7 Phase 3

Why ELM1 Cannot Learn this Algorithm

The system ELM1 is not able to learn this method of division. Let us examine phase 1 of the solution to see why. We notice that in step 3 the system was dealing with the goals

$$0+4 = X2 \quad \& \quad X2/2 = X1.$$

The object of the next few steps was to transform the first of the two goals to obtain

$$2+2 = X2 \quad \& \quad X2/2 = X1.$$

This is achieved by repeatedly subtracting "1" from the second integer and then adding it to the first integer. The process has to stop when the first integer becomes equal to the divisor:

```
2 + 2 = X2  &   X2 / 2 = X1.
|   |           |
|   2nd integer |
|               |
1st integer     divisor
```

Then clause 'eq' has to be applied. Fig. 5.5 shows what happens when this is done.

We see that to terminate phase 1 the system has to be able to examine the first two goals to be solved. However, system ELM1 cannot do that and this is why it cannot learn this algorithm.

The system ELM2, however, can learn this algorithm without any difficulties. In the following we shall show how this was done.

Error Correction

Several different errors were detected in the course of solving the problem of division 4/2=X1. We shall now describe how some of these errors were corrected.

The following figure shows the 'error information' which has been stored during the search phase on the basis of which the errors were corrected.

Phase	Step	Current Goal Stack	Clause Selected	Error
1	1	4/2=X1 & write(X1)	subsd	SSE
	2	4=X2 & (X2/2=X1 & write(X1))	izero	CSE
	13	2+2=X2 & (X2/2=X1 & write(X1))	eq	CSE
2	14	(2+2/2)=X1 & write(X1)	distd	SSE
	15	(2/2+2/2)=X1 & write(X1)	subs	CSE
	16	2/2=X2 & (X2+2/2=X1 & write(X1))	cancd	SSE
	17	1=X2 & (X2+2/2=X1 & write(X1))	eq	CSE
3	19	2/2=X2 & (1+X2=X1 & write(X1))	cancd	SSE
	20	1=X2 & (1+X2=X1 & write(X1))	eq	CSE

Fig. 5.8 Error Information Stored

Error from Step 1

The error detected in step 1 is a simple selection error. This error was corrected by the addition of the orderings 'subsd > izero' and 'subsd > eq'.

Error from Step 2

The error in this step arose because clause 'pred' was selected instead of clause 'izero'. If the system tried to correct this error by the addition of priority orderings only, a conflicting system of priority orderings would have been obtained, as shown in the following figure.

```
suc > subs1 >  asoc  >  subz > subs        )
                        subz > eq          )    Current
                        pred > eq          )    priority
                                           )    orderings
                        subz > izero       )
                        pred > izero       )
                       subsd > izero       )
                       subsd > eq          )

                                           )    Conflicting
                       izero > pred        )     ordering
```

Fig. 5.9 Priority Orderings Used

The error mentioned was corrected by conflict resolution and one new clause was produced as a result of that. This clause was called 'izero1'. The following selection and rejection contexts were used in the process:

```
Sel. context:  4=X2 & (X2/2=X1 & write(X1))        Assoc. Ordering:
Rej. context:  2=X2 & (3+X2=X1 & write(X1))          pred > izero
```

The new clause generated is shown below. It contains two new constraints, each referring to the second goal on the goal stack. We see that this goal should be in the form of an equation with '../..' on the left hand side. The new clause was given priority over clause 'pred'.

```
izero1:  X1=X2 & Gs   <-  ( X1=:4 v
                            Gs=:((../..=..) & ..)  v
                            Gs=:((X2/..=..) & ..))
                            & !  & 0+X1=X2
```

Error from Step 13

The error in this step arose because clauses 'subz' and 'izero' were incorrectly selected instead of clause 'eq'. The error was corrected by conflict resolution. The following figure shows the priority orderings as they were before this error was corrected.

```
suc > subs1 > asoc > subz                    > subs     )
                     subz                    > eq       )   Current
                                       pred  > eq       )   priority
                                                         )   orderings
                              izero1 > pred  > izero    )
                                       subsd > izero    )
                                       subsd > eq       )

                                         eq  > izero1   )   Conflicting
                                         eq  > subz     )   orderings
```

Fig. 5.10 Priority Orderings Used

The error detected in this step was corrected by generating a new constrained version of clause 'eq'. The following selection and rejection contexts were used in the process:

Sel.context:	2+2=X2 & (X2/2=X1 & write(X1))	Assoc. Ordering:
Rej.context:	3+2=X1 & write(X1)	subz > eq
	2=X2 & (3+2=X1 & write(X1))	pred > eq
	4=X2 & (X2/2=X1 & write(X1))	izero1 > pred

The rejection contexts were used one by one in conjunction with the selection context shown. The following clause was generated as a result:

```
eq1:  X1=X1 & Gs <- (X1=:(2+..)   v  X1=:(X2+X2)  v
                     X1=:(X3+..)  &  Gs=:((../X3=..) & ..) v
                     X1=:(..+X4)  &  Gs=:((../X4=..) & ..) ) & !
```

The new clause was given priority over clauses 'subz' and 'izero1'.

Correction of the Remaining Errors

All the other errors were corrected in a similar way. To correct the error detected in step 15, for example, clause 'subs2' was generated by the system. The following contexts were used in the process:

```
Sel.context:  2/2+2/2 =X1  &  write(X1)          Assoc. Ordering:
Rej.context:  3+2=X1       &  write(X1)             subz > subs
```

The new clause generated was as follows:

```
subs2: X1+X2 =X3  &  Gs   <-  X1=:(../..)  v  X2=:(../..)  & ! &
                              X1=X4        &  X4+X2 =X3
```

To correct the error detected in step 17 the system generated another version of clause 'eq', called 'eq2'. The following selection and rejection contexts were used in the process:

```
Sel.cont.:  1=X2 & (X2+2/2 =X1 & write(X1))      Assoc. Ordering:
Rej.cont.:  2=X2 & (3+X2 =X1   & write(X1))          pred > eq
```

The new clause generated was as follows:

```
eq2:   X1=X1      &  Gs  <-  (X1=:1 v
                              Gs=:((X1+..= ..) & ..)  v
                              Gs=:((..+(../..)= ..) & ..)) & !
```

The error detected in step 20 did not have to be corrected because it was found that clause 'eq2' was correctly selected in the associated context. Clause 'eq2' was, however, modified using the method of 'learning from examples' described before. The modified version of this clause is as follows:

```
eq2:  X1=X1  & Gs  <-  X1=:1 v Gs=:((X1+..= ..) & ..) & !
```

The priority orderings obtained are shown in the following figure.

```
suc > subs1 >  asoc > subz                    > subs
                      subz                    > eq
                                        pred  > eq
          eq1 > subs2 > subz
          eq1 >                 izero1 > pred > izero
                                         subsd > izero
                                         subsd > eq
                                eq2    >  pred
```

Fig. 5.11 Priority Orderings Obtained

Re-run of the Same Problem

The same problem was given to the system again to find if any errors would occur. We found that two simple selection errors and one conflicting selection error were made by the system. They were corrected by the system in a normal way. Let us see why these errors occurred.

The first simple selection error occurred in step 3. Clause 'izero1' was incorrectly selected, together with clause 'subz'.

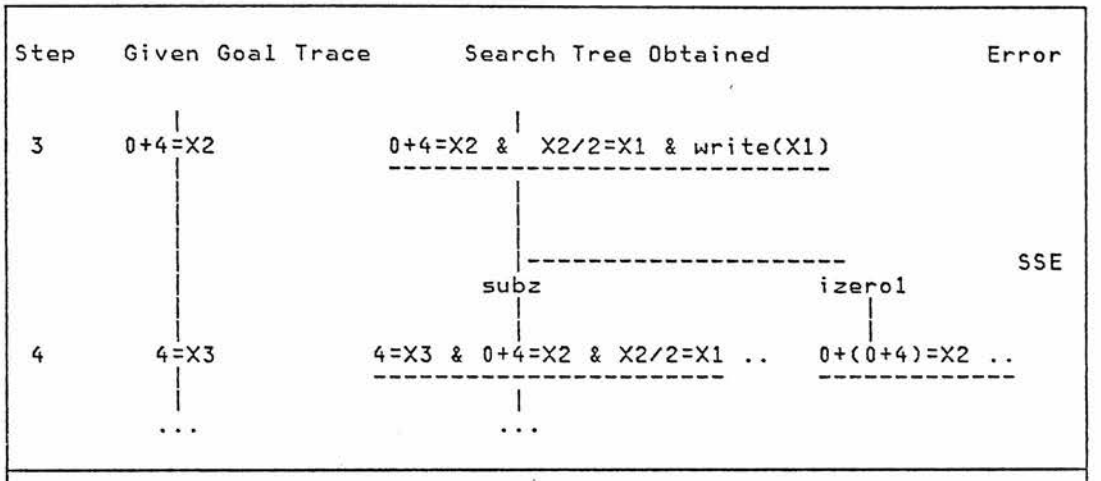


Fig. 5.12 The First Error During the Re-run

This error arose because when clause 'izero1' was generated the ordering 'subz > izero1' was not introduced by the system. The system considered the introduction of this ordering, since the ordering 'subz > izero' existed at that time (see chapter 3.9). The system performed selection in the context associated with this ordering, but as no error was detected the system assumed that the ordering 'subz > izero1' was not really needed. We see that this assumption was wrong.

The second simple selection occurred for similar reasons. The following figure shows in which step the error occurred and which clauses were selected.

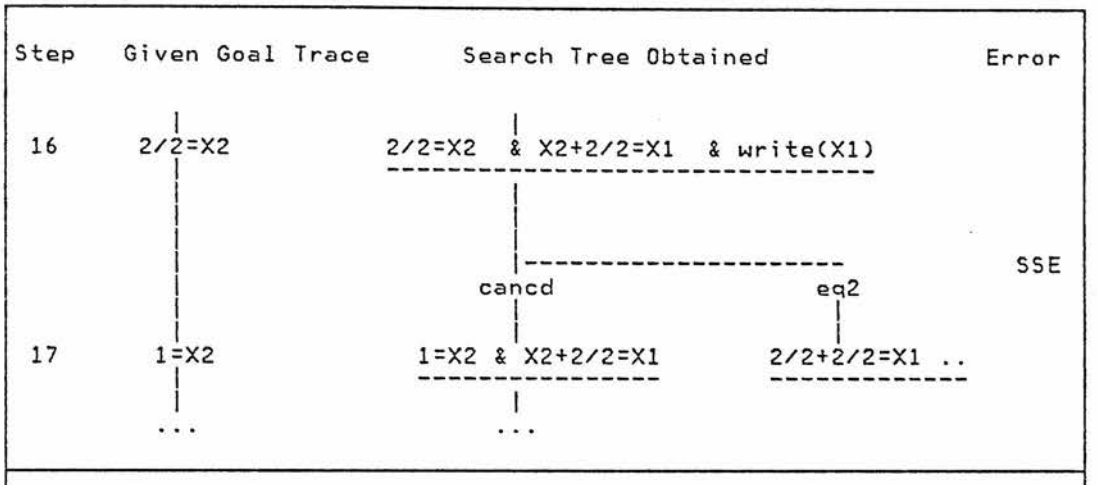


Fig. 5.13 The Second Error During the Re-run

The second simple selection error was corrected by the introduction of the ordering 'cancd > eq2'.

The following figure shows when the third error occurred during the re-run.

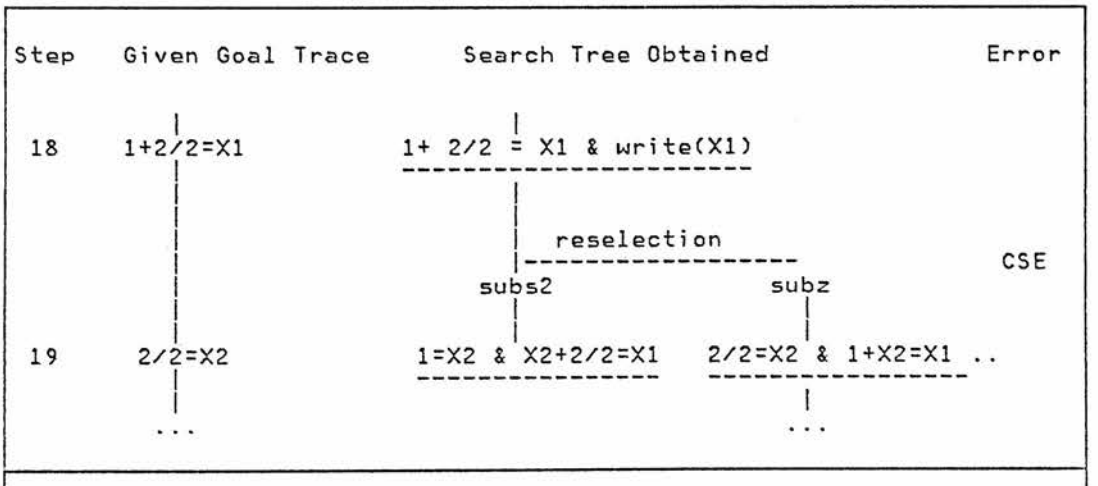


Fig. 5.14 The Third Error During the Re-run

Let us examine why this conflicting selection error has occurred. We notice that no error was detected in step 18, when this problem was solved the first time, and because of that, the selection context associated with this step was not stored. Had this context been stored and taken into account when errors were being corrected, one of the clauses generated would have been differently constrained (clause 'subs2') and this error would have occurred during the re-run.

The error mentioned before was corrected by conflict resolution. The following selection and rejection contexts were used:

```
Sel.context:  1+2/2 =X1  & write(X1)          Assoc. Ordering:
Rej.context:  2/2+2/2=X1 & write(X1)            subs2 > subz
```

This clause was generated:

```
subz1:  X1+X2 =X3  &  Gs  <-  int(X1)  & !  & X2=X4  & X1+X4=X3
```

The new clause was given priority over clause 'subs2'. Priority ordering 'eq1 > subz1' was also introduced because the ordering 'eq1 > subz' was found to exist, and an error would occur without it, if one of the old subgoals was solved again. Introduction of these additional priority orderings was discussed in chapter 3.9. The priority orderings obtained are shown in the following figure.

```
suc > subs1 > asoc  >  subz                        > subs
                       subz                        > eq
                                                pred > eq

eq1 > subz1 > subs2 >  subz  > izero1 >  pred > izero
                             distd > subsd > izero

                       cancd >  eq2   >  pred
                       cancd          > subsd > eq
```

Fig. 5.15 Priority Orderings Obtained

Another Problem of Division

The next problem dealt with was '9/3=X1'. No errors occurred during the solution of this problem.

Two of the existing clauses (izero1,eq1) were modified, however, on the basis of 'learning from examples'. The method was discussed in chapter 3.11. The modified clauses are shown in the following.

```
izero1:  X1=X2   & Gs  <-  Gs=:( ../..=.. & ..)    & !  & 0+X1=X2
eq1:     X1=X1   & Gs  <-  X1=:X3+..  &  Gs=:(../X3=.. & ..) & !
```

What the System has Learned

Several new clauses which were acquired by the system were shown together with the associated priority orderings. Let us see how we can interpret the results that have been obtained.

How to start the process of division

If 'X1' is to be divided by 'X2' then the dividend should be replaced by a suitable term which we have to find. This rule is expressed by the clause 'subsd' which has been given to the system:

```
subsd: X1/X2=X3 & Gs  <-  ! & X1=X4 & X4/X2=X3
```

If the clause shown can be used there is no point in trying to apply clause 'eq' which would simply give 'X1/X2' as the answer. Also, there is no point in modifying the dividend by adding '0' to it. That is the orderings

```
subsd > eq
subsd > izero
```

should be respected.

What to do with the dividend

If problem 'X1=X2' has been encountered in the course of dividing two integers then '0' should be added to 'X1', as clause 'izero1' shows:

```
izero1:  X1=X2 & Gs  <-  Gs=:(../..=.. & ..)  & !  & 0+X1=X2
```

Notice that the constraint of this clause which has been generated by the system ensures that this clause is selected in the right context only, that is when division is being performed.

It should be remembered that if '0' can be added to 'X1' then there is no point in trying to find the predecessor of 'X1'. That is the ordering 'izero1 > pred' should be respected.

How to transform the dividend further

Any of clauses which are used in the process of calculating integer addition should be used preferentially before trying to use clause 'izero1' described before. That is, the priority orderings

```
suc > subs1 > asoc > subz > izero1
```

should be respected. Without this rule it may be difficult to complete the phase 1 of the process of division. Without this rule clause 'izero1', for example, could be repeatedly invoked and '0' repeatedly added to the expression dealt with. That is, '0+X1' could be transformed into '0+(0+X1)' and so on.

When to stop transforming the dividend

If division is being performed and if the divisor is 'X3' then the current goal should be examined. If it is of the form 'X3+X4=X1' then the current goal should be regarded as solved and 'X1' should be replaced by 'X3+X4' wherever possible. That is, clause 'eq1' should be used:

```
eq1:  X1=X1 & Gs  <-  X1=:X3+X4  &  Gs=:(../X3=.. & ..) & !.
```

Notice how the constraints generated by the system restrict the use of this clause.

It should be noted that the operation described should be performed preferentially before we try to modify 'X3' or 'X4'. That is the priority orderings

```
eq1 > subz1 > subs2 > subz
```

should be respected. The last rule mentioned determines when we should stop transforming the expression 'X3+X4=X1' and complete phase 1 of the process of division.

What to do if the dividend is a sum

If a sum of two numbers is to be divided by number 'X' then each of the two numbers should be divided by 'X' and the results should be added together. That is clause 'distd' should be used under such circumstances.

```
distd: (X1+X2)/X3=X4 & Gs <- ! & X1/X3+X2/X3=X4.
```

The sum 'X1+X2' itself should not be modified. That is the ordering 'distd > subsd' should be respected.

How to do a sum and a division at the same time

If 'X1' is to be added to 'X2' and if 'X1' is of the form 'X5/X6' then 'X5/X6' should be calculated first, and the addition should be worked out afterwards. That is clause 'subs2' should be used under such circumstances:

```
subs2: X1+X2=X3 & Gs <- (X1=:X5/X6 v X2=:X7/X8) & ! &
                        X1=X4  &  X4+X2=X3.
```

If 'X2' is of the form X7/X8 then clause 'subs2' should not be used even though the constraint 'X2=:X7/X8' is actually true. Clause 'subz1' should be used respecting the ordering 'subz1 > subs2'.

When division is trivial

If some number is to be divided by itself then the result is equal to '1'. Clause 'cancd' should be used under such circumstances:

```
cancd:  X1/X1=X2  &  Gs  <-  1=X2.
```

Clause 'cancd' should be used whenever possible; in particular before we try to use clause 'subsd' shown before. That is the priority ordering 'cancd > subsd' generated by the system should be respected.

How to use a partial result

If the current goal is of the form 'X1=X1' and if 'X1' is equal to '1' then the current goal should be regarded as solved, and 'X1' should be replaced by '1'.

Similarly, the current goal should also be regarded as solved if the

second goal contains a sum of 'X1' and some other number. Clause 'eq2' should be used under such circumstances.

```
eq2:  X1=X1 & Gs  <-  X1=:1  v  Gs=:(X1+..=.. & ..) & !.
```

Clause 'eq2' should be used only if it is not possible to simplify the expression dealt in any way, but if this clause can be used there is no point in trying to find the predecessor of 'X1'. That is the orderings

```
cancd > eq2 > pred
```

generated by the system should be respected.

5.7 SUMMARY

In this chapter we have described our extended system ELM2. In this system we are able to control selection of clauses more effectively.

The clauses used are rather different from the clauses used in ELM1. The clause head may contain a number of predicates and each clause can be selected only if the clause head matches the current goal stack. Thus the selection of these clauses is affected by which goals are currently on the goal stack. The constraints affect selection, too, because they may refer to any subterm in the clause head.

Generation of Constraints

New constraints are generated on the basis of 'selection' and 'rejection' contexts, as in ELM1. However, the contexts used in ELM2 include the whole goal stack. Apart from this one difference new constraints are obtained as in ELM1.

The system ELM2 could easily be extended so that it would be able to reorder the existing subgoals. We have pointed out that this would be quite useful, since often the difficulty of solving a particular set of subgoals depends on which subgoal is solved first.

Experimental Results

Two sets of problems were studied in detail to verify that the techniques described could be applied in practice. The problems dealt with addition of integers and division of integers.

A number of conflicting selection errors were dealt and a number of new clauses were generated as a result. In the example dealing with division of integers the constraints of most of the new clauses referred to the second goal on the goal stack. The corresponding errors could not be corrected by ELM1, because only the first goal on the goal stack would have been taken notice of at any given time, and the errors would keep recurring.

The system ELM2, however, deals with all errors succesfully. After all errors have been corrected the system is capable of solving other problems of integer division without any difficulties.

6 RELATION TO OTHER WORK

6.1 WINSTON: LEARNING STRUCTURAL DESCRIPTIONS

A number of researchers have tried to find out how 'concepts' get acquired from examples. Plotkin (1969), for example, has examined the process of generalization and shown what role it plays in learning. Vere (1976) has applied generalization to the acquisition of 'actions'. His system can learn, for example, particular chess moves if it is given sufficient number of examples showing where the pieces are before and after each move.

Winston has shown how models of various concepts can be acquired and used in the process identifying objects in the given scene. Winston's system can learn what simple block structures in the form of houses or arches, for example, look like.

The models of concepts acquired show which relationships are important. Winston's model of a 'house', for example, shows that it must consist of a 'wedge' and a 'block'. Moreover, the 'wedge' must be supported by the 'block'. Any two objects which will satisfy this description will qualify as a 'house'.

Winston maintains that learning is not very effective if the

system is shown only particular instances of concepts, that is examples of, say, houses or arches, because the system cannot easily work out which relationships are mandatory and which are only incidental.

Winston shows the usefulness of so called near-miss examples. These fail to be examples of the particular concept because one (or several) relationships are not satisfied. Carefully selected 'near-misses' allow the system to identify those relations which are important for the recognition of a particular concept.

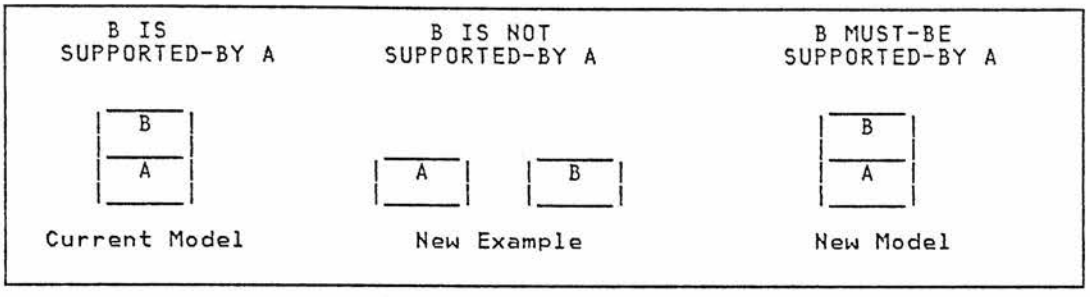
Elementary Model Building

Let us see how the model of some concept is generated in Winston's system. The first example of a concept encountered is always used as the initial model of that concept. The model is updated if it does not agree with other examples and 'near-miss' examples presented later.

If the model of a concept contains some predicate which is not satisfied in the 'near-miss' example, the existing model is modified to indicate that this predicate MUST BE satisfied (*).

If, for example, one block is supporting another in the model, and if the two blocks are not supported in the 'near-miss' example, the model is amended by adding 'MUST-BE' to the predicate 'SUPPORTED-BY'.

(*) Winston does not use the term 'predicate' since all relationships are represented as 'networks' with 'pointers'. Section 'The supplementary-pointer C-note' (p.138) explains how models are updated on the basis of 'near-miss' examples.



What Our System Does

Let us see what our system does in similar circumstances. In our system the existing 'models of concepts' are represented in the form of clauses which are modified on the basis of 'selection' and 'rejection' contexts. These contexts serve a similar role as the 'examples of concepts' and the 'near-miss' examples used by Winston.

Our method for updating clauses differs somewhat from the method Winston uses for updating the model. We do not try add the prefix MUST-BE to any of the existing predicates. The following figure shows how the two systems compare.

Winston's System	Our System
Information Used:	Information Used:
Current Model Near Miss Example	Current Clause Selection Context Rejection Context
Action:	Action:
Add 'MUST-BE' to the existing predicates in the current model	Add new predicates to the existing predicates in the clause dealt with

Clauses in our system are modified by the introduction of

new predicates called 'constraints'. All predicates chosen must necessarily be satisfied in the selection context since this is one of the conditions which is tested by the system (see Chapter 3.4). The predicates added must also be satisfied in any context in which this clause is to be used because otherwise the clause would not have been selected. This is why we do not really need to use the prefix 'MUST-BE' in our system.

Our method of modifying the existing clause (model) is more powerful than Winston's. In our system any predicate may be added to the existing predicates in the clause provided it is 'true' or 'false' in the appropriate contexts. All Winston's system can do is add the prefix 'MUST-BE' to the existing predicates in the model in these circumstances.

Our system has other limitations, however. Winston's type of concepts are most naturally described using sets of predicates. The order in which these predicates appear is unimportant. In our system, however, all predicates are dealt with in the order in which they appear. If both the selection and the rejection context consisted of several predicates then our system would not be able to to generate constraints correctly, because, for example, P1&P2 and P2&P1 would appear different. In this respect our system is more limited than Winston's.

Dealing with Different Alternatives

If we do what Winston's system does and compare the model of some concept with the 'near-miss' example, any number of predicates may be found which appear in the model and which are not satisfied in the example. We do not know which of the predicates should be used in updating the model.

In Winston' system the problem is dealt with in the following way. One of the predicates is chosen and 'MUST-BE' is added to it and the other predicates are considered irrelevant for the time being. However, if it is found that the predicate chosen is not satisfied with another example of the same concept the system assumes that an incorrect choice was made before. The system 'backtracks' to explore one of the other possible alternatives. The system backtracks level by level and so it may take some time before the mistake is actually corrected (*).

Winston himself is critical of this method. During backtracking the system is blindly exploring all possible paths on the way back to where the incorrect decision was made. Winston points out that it would have been better if the system could 'jump' directly back to where the problem began.

What Our System Does

In our system, too, more than one predicate can often be found which could be used to modify the clause dealt with. However, rather than exploring one alternative at a time, an expression is generated by the system containing all the alternatives explicitly represented. The expression is modified upon encountering new examples of the concept dealt with and the search tree is, in effect, pruned down. This is why our method of dealing with different alternatives is better than Winston's.

(*) See the discussion on 'Multiple C-notes', p.147.

Modification of Existing Models

In Winston's system predicates may be deleted from the existing model under certain circumstances. This happens if a predicate is found in the model which is not satisfied with the next example encountered and if the model does not say that this predicate should be satisfied (that is if MUST-BE was not added to it).

Under certain circumstances the predicate in the model may also be replaced by another one if it happens to be satisfied with the current example. For example, the predicate 'A-Kind-of-Brick' is replaced by 'A-Kind-of-Object'. We notice that the second predicate is more general than the first one and his 'network' in which all data is stored shows that (*).

What Our System Does

We have mentioned that Winston's aim is to delete or replace the predicates which do not have the prefix MUST-BE. Winston is, in effect, trying to delete or replace predicates which formed the description of the examples of concepts (selection contexts) presented before. This is quite different from what we do under these circumstances.

What we try to do is modify the clause selected (the model). We have mentioned before that our aim is to eliminate all the alternatives which are not satisfied in the new context. We have also explained what the benefits of this are.

No attempt is made in our system to modify the selection

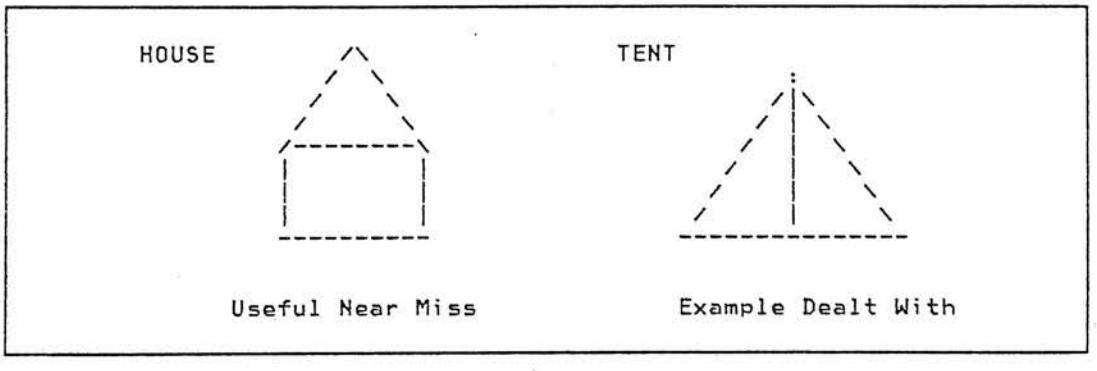
(*) See the discussion on 'The A-kind-of-merge C-note', p.135.

contexts (examples of concepts) once they have been stored. Perhaps it would be useful to modify our system so that it would do what Winston's system does. The description of selection contexts stored could be modified whenever new selection contexts have been encountered. More work is needed to establish how exactly this should be done and whether we would gain anything as a result.

Use of Examples as Near Misses

In Winston's system models of various concepts are acquired one after another. For example, after the system has learnt what a 'house' is, the system is taught what a 'tent' looks like. However, no check is ever made whether the current model would not fit the description of the previous examples. What would be the value of that? Well, the examples of various concepts can under certain circumstances be used as 'near-misses' instead of the 'near-miss' examples supplied.

We notice that Winston's example of a 'house', for example, is actually quite similar to the example of a 'tent', because both examples consist of two objects and one of them must be in the shape of a 'wedge'. Also, both objects must touch each other (they must satisfy the relation 'marry'). We see that the example of the 'house' could be used as a 'near miss' when we are trying to learn what a 'tent' looks like.



What Our System Does

Our system tries to do what we have just described. It tries to use certain examples of old concepts as 'near-miss' examples when refining the current model. If, for example, our system was trying to refine the model of a 'tent', the example of a 'house' presented before could be used as a 'near-miss' in this case.

The system itself tries to determine which of the examples presented before will be used as 'near-misses'. That is the teacher does not have to specify that. In Winston's system all 'near-misses' have to be explicitly provided. We see that in this respect our system is somewhat less dependent on outside help than Winston's system. Our system is really taught to solve problems and while it is doing this, it learns to differentiate between different concepts.

6.2 BIERMANN: INFERENCE OF PROGRAMS FROM EXAMPLES

Many people have thought that it would be nice if programs could be generated in some automatic way from given input and output. Several researchers undertook the task of exploring how this might be done. Among them were, for example, Hardy (1975), Jouannaud & Kodratoff (1978) and Biermann whose work we chose to describe here and relate to our own (*).

Generation of programs from given input and output is a very difficult task in general, because the given input and output values does not give us much information about how the program should be constructed. This is why most of the researchers looked at rather special class of problems. Many of the problems they looked dealt with 'lists'. The given input and output shows not only what the program should produce, but also how this might be done.

Suppose the input is, for example, '((A.B).C)' and the desired output is '(C.(B.A))'. We can see that the first element of the input list appears as the last element in the output. The relationship between the two elements suggests that if we want to obtain the output the first input element has to be put at the end. Similar relationships exist among the other elements as well.

Biermann points out that the structural information showing how the output should be formed from the input is very valuable when we are trying to synthesize the desired program. Let us now see how Biermann's system works.

Biermann's system works in distinct two phases. In the first phase the system tries to find out how the output can be formed

---

(*) The full title of Biermann's paper discussed here is "The Inference of LISP Programs from Examples".

from the input. The result appears in the form of a computation trace which is used in the next phase to construct the desired program. Let us first see how computation traces are generated in Biermann's system.

Generation of Computation Traces

Computation traces are generated from the given input and output. The system tries to find out how the output can be formed from the input with a certain number of primitive functions. Biermann uses four such functions: (*)

```
IDENTITY:          (Fi X) = X
CAR:               (Fi X) = (Fj (CAR X))
CDR:               (Fi X) = (Fj (CDR X))
CONS:              (Fi X) = (CONS (Fj X) (Fk X))
```

Fig. 6.1 Primitive Functions Used by Biermann

Many different functions can be constructed from the primitive functions shown before. For example, the set of primitive functions which is shown in the following figure will transform the list X=((A.B).C) into the list Y=(C.(B.A)).

(*) CAR, CDR and CONS are primitives of LISP. The function CAR returns the left part of the s-expression (symbolic expression) supplied to it and CDR returns the right part. If the s-expression is (A.B), for example, CAR will return A, and CDR will return B. If the s-expression is an atom (primitive constant of LISP) CAR and CDR return NIL. CONS produces an s-expression from two s-expressions supplied to it. More details can be found in LISP 1.5 Programmer's Manual (McCarthy et al., 1962).

```
(F1 X) =  (CONS (F2 X) (F3 X))
(F2 X) =  (F4 (CDR X))
(F3 X) =  (CONS (F5 X) (F6 X))      <----
                                         | CONS is used
(F4 X) =   X                             | before  CAR
                                         |
(F5 X) =  (F7  (CAR X))             <----
(F6 X) =  (F8  (CAR X))

(F7 X) =  (F9  (CDR X))
(F8 X) =  (F10 (CAR X))

(F9 X) =   X
(F10 X) =  X
```

Fig. 6.2 A Particular Set of Primitive Functions

In general different sets of primitive functions may be found that will transform the input into the same output. The set of primitive functions which is shown in the following figure, for example, perform the same function as the primitive functions which we have shown before. (They will transform X=((A.B).C) into Y=(C.(B.A))).

```
(F1 X) =  (CONS (F2 X) (F3 X))
(F2 X) =  (F4 (CDR X))
(F3 X) =  (F5 (CAR X))              <---
                                         | CAR is used
(F4 X) =   X                             | before CONS
                                         |
(F5 X) =  (CONS (F6 X) (F7 X))      <---
(F6 X) =  (F8  (CDR X))
(F7 X) =  (F9  (CAR X))

(F8 X) =   X
(F9 X) =   X
```

Fig. 6.3 Another Set of Primitive Functions

The two sets of primitive functions which we have shown differ in the following way. In the first set the CONS operation is applied before the CAR operation. In the second set the order of applying CONS and CAR is reversed.

Biermann maintains that if there is a choice between applying a CAR or CDR operation to X or applying CONS to build Y, then one should always choose the CAR or CDR operation. If

we do that then the system will generate regular LISP programs which have certain desirable properties (see Biermann's paper).

Computation traces are sets of primitive functions obtained on the basis of the given input and output in a certain systematic way. They show which primitive operations should be applied when and usually they are represented graphically in the form of 'trees'. The following figure, for example, shows the primitive functions from Fig. 6.3 rearranged in this manner.

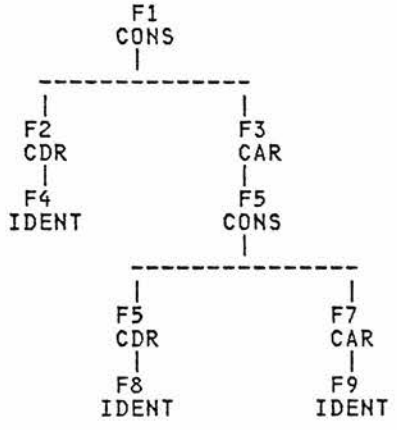


Fig. 6.4 An Example of a Computation Trace

The procedure that constructs computation traces is referred to as t(X,Y) (X represents the given input and Y is the desired output). The procedure t(..) may be invoked recursively with subterms of X and Y. On each invocation the procedure determines which one of the four primitive operations (CONS, CAR, CDR and identity) should be done just then. A detailed description of when each particular primitive function is to be used can be found in Biermann's paper.

Program Synthesis

After the computation trace has been constructed and a set of primitive functions obtained, Biermann's system tries to synthesize the desired program. This is achieved by merging some of the primitive functions obtained before and also by the addition of LISP conditionals where appropriate. First let us see what merging is.

Sometimes it is possible to replace a number of functions by a single function without affecting what the program does to the input. The process of identifying two or more functions and replacing them by one is referred to as 'merging'.

It is not difficult to see which functions we could try to 'merge'. They are the ones that contain the same type of operation (such as CONS). Two functions that could be merged are shown below.

```
(F1 X) =  (CONS (F2 X) (F3 X))
(F5 X) =  (CONS (F6 X) (F7 X))
```

If we want to merge two or more functions that contain different operations then we have to use a conditional which allows us to specify the conditions under which each function should be invoked. For example, suppose that we want to convert

```
(F4 X) =   X
(F1 X) =  (CONS (F2 X) (F3 X))
```

into

```
(F1 X) =   X
(F1 X) =  (CONS (F2 X) (F3 X))
```

and we want to specify that the first function should be invoked if X is an atom, such as 'a' or 'b' or 'NIL'. The following conditional, for example, achieves just that.

```
(F1 X) = (COND   ( (ATOM X)  X )
                 ( T         (CONS (F2 X) (F3 X))))
```

Conditionals used by Biermann may contain any number of predicate-function pairs:

```
(COND   ( p1      f1   )          where
        ( p2      f2   )          p1 - pn-1 ... predicates
           ...                    f1 - fn   ... functions.
        ( pn-1    fn-1 )
        ( T       fn   )),
```

The predicates are tested one by one and if one is found that is true, the corresponding function is invoked and the remaining predicates are ignored. The last predicate tested (T) is always true and so the function 'fn' is invoked unconditionally if all the predicates tested before happen to be false.

Let us now see how a particular set of primitive functions can be transformed by 'merging' and by the introduction of 'conditionals'. Let us consider the primitive functions which were already shown before (in Fig. 6.3). If these were given to Biermann's system a number of functions would have been merged and one 'conditional' would have been added. The following figure shows which functions would actually be merged and also what the resulting program looks like.

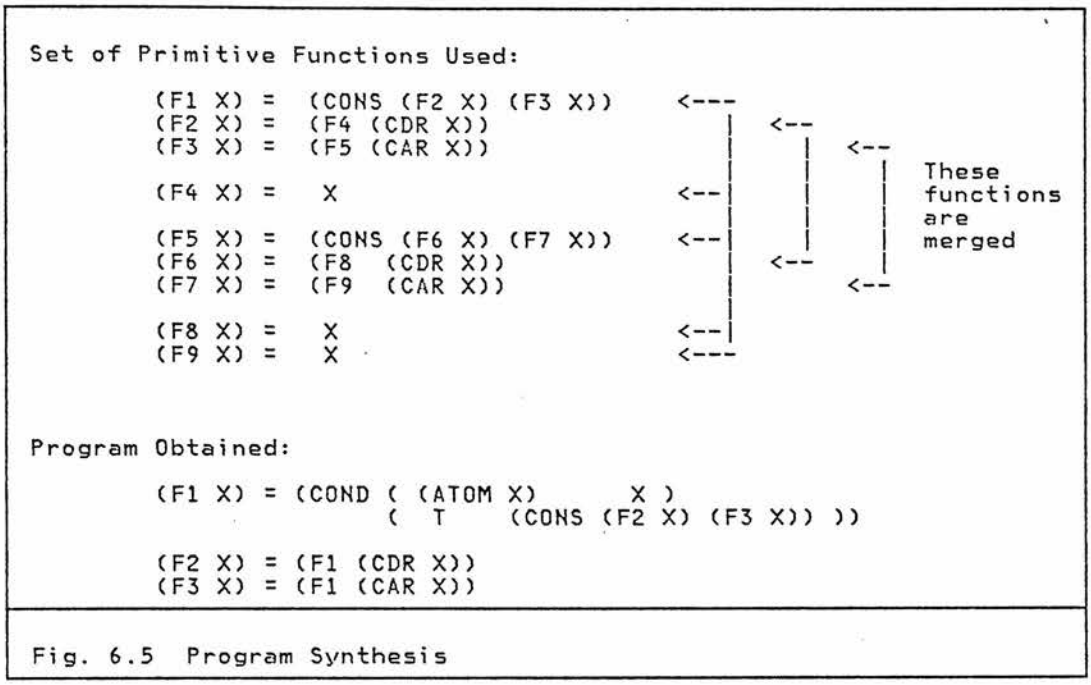


Fig. 6.5 Program Synthesis

The algorithm for deciding which functions should be merged and where conditionals should be used is really quite complex. We have to make sure that the program can in the end do what it is supposed to do and that is produce the desired output Y from the given input X. The computation trace constructed before is carefully processed so as to minimize the search.

The computation trace is split up into two parts by a frontier which indicates how much of the trace has been processed at any given time. The following figure shows how the trace shown before was split up at a certain stage.

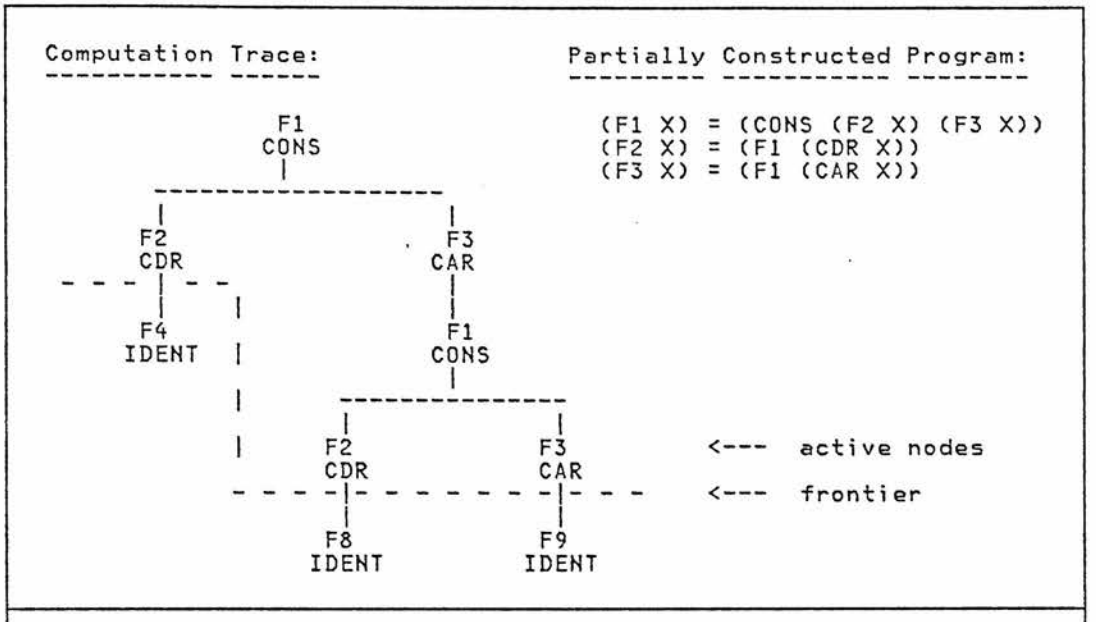


Fig. 6.6 Partially Processed Computation Trace

The synthesis procedure chooses 'active nodes' along the frontier and attempts to advance the frontier and, if necessary, perform merging. How exactly this is done is explained in Biermann's paper.

Comparison of Our System with Biermann's

In the following we shall describe in what way our system is similar to Biermann's and how the two systems differ.

Common Two Phase Approach

We notice that Biermann's system and our system, too, work in two phases. First, each system tries to determine which operations will transform the given input X into Y, or in our case, solve the given problem. Then after this has been done each system tries to generate a program which serves as a more

compact and more general representation for all the operations needed.

Amount of Information Stored

We notice that our system differs from Biermann's in the amount of information stored during the search for a solution of the given problem.

In Biermann's system the whole computation trace is produced and used while the new program is being generated. Our system, on the other hand, is built on the assumption that if the desired trace can be obtained with the help of the existing program than there is no point in storing it. The information which is stored, however, represents that what the system did not expect to happen. The stored data is referred to as 'error information' (see Chapter 3).

Our System Need Help During Search

Our system requires more help than Biermann's when it is trying to solve the given problems and find out which clauses should be used when. In the following we shall explain why this is so.

The problems dealt with by Biermann are somewhat special. If we inspect the given input and the output it is not too difficult to see how the elements of the input can be rearranged and how the output can be obtained. This is also why Biermann's system does not have any difficulties in identifying the set operations that transform the input in the desired way.

The problems dealt with by our system are quite different from Biermann's. The problems themselves do not give us much information about how we should solve them. For example, suppose that we are dealing with equation

$$(X1 + 3) + 1 = 7$$

and we know that the solution is X1=3. It is difficult to guess how the solution should be obtained without knowing anything about equations. Even if we knew how equations could be transformed from one form to another we might still have difficulties in obtaining the solution. In general there may be several rules that could be applied at any given moment, and so the search space that has to be explored can be quite large. This is why we have decided to help our system in the search.

We have explained before that the system is shown how the given equation is to be transformed step by step. It is presented with the sequence of 'goals' that the system should generate. The given goals (goal trace) enable our system to find which operation should be performed when. The given 'goal trace' is a richer source of information than the input and output used by Biermann.

Our System is Unable to Rename Predicates

Our system cannot learn to solve a certain class of problems dealt with by Biermann. In the following we shall explain why this is so.

In our system new clauses are generated by the modification of the existing clauses. In general new clauses are generated by adding new predicates to the existing predicates in the clause dealt with, and it is unable to modify them afterwards.

For example, it is unable to generate the clauses

```
car1:    f1 (X1.X2, X3)  <- !  & f2 (X1,X3)
cdr1:    f2 (X1.X2, X3)  <- !  & f1 (X2,X3)
```

from

```
car:     f1 (X1.X2, X3)  <- !  & f1 (X1,X3)
cdr:     f1 (X1.X2, X3)  <- !  & f1 (X2,X3) (*).
```

We notice that clauses 'car1', for example, can be obtained from 'car' by changing some of the predicate names, that is in this case by changing 'f1' into 'f2'. This is what our system cannot do, however, and because of that it cannot solve some of the problems that Biermann's system is able to solve.

This is wording of Biermann's Example 5: Apply the operations CDR, CDR, CAR, CDR, CAR, CDR etc. to the input until an atom has been found and then return that atom.

We see that after the CDR operation has been applied once the system is supposed to apply pairs of CDR and CAR operations until an atom is reached.

Only those systems which can in one way or another remember what the last operation was can deal with the previous problem (and other similar problems). Our system cannot do that. Biermann's system is able to deal with the previous problem because it can choose different names for various functions.

The use of two different function names (F1,F2) in the following example guarantees that the CDR operation is applied after CAR, and that the CAR operation is applied after CDR. A similar program can be generated by Biermann's system.

(*) Each clause shown can be converted into LISP without difficulties. This is what clause 'car1', for example, will look like after it has been translated into LISP: (F1 X) = (F2 (CAR X)).

```
(F1 X) = (COND ( (ATOM X)          X )
               ( T       (F2 (CDR X))))

(F2 X) = (COND ( (ATOM X)          X )
               ( T       (F1 (CAR X))))
```

The problems that we have considered were quite different from Biermann's. Our system could always determine what should be done next from the expression dealt with. Our system did not have to remember which operation was performed when.

Our system could be extended so that it could deal with the problem mentioned and other problems which are similar to it. The notion of the 'context' could be extended to include the representation of various operations performed in the past. The differences between the selection and rejection contexts could serve as the basis for the generation of new constraints. These would not only refer to the expressions dealt with but also to the operations performed in the past.

Introduction of New Predicates

Biermann has paid little attention to the problem of which predicates should be used in the 'conditional operators' that can be introduced by his system. We believe that our system is more sophisticated than Biermann's in the way it can choose predicates and constrain the invocation of clauses. Let us see first how the predicates are generated in Biermann's system.

Biermann points out that every time the program is changed in any way, that is, for example, by 'merging', the predicate generation routine must discover whether the predicates can be found and it must produce them if they exist.

Biermann takes the partially constructed program and tries to execute it noting all the s-expressions obtained in the

process. The predicates are chosen so that they would have the correct truth-value with the s-expressions noted (*).

The basis of Biermann's method for generating new predicates seems to be similar to the method we use in our system. Certain s-expressions which we call 'contexts' are used to test whether a particular predicate can be used to constrain the invocation of a particular function (clause).

Biermann's explanation of this process is very brief, however. He does not mention many of the problems which we discuss in our thesis in detail. Biermann does not mention, for example, that if care is not taken the predicates obtained may be either 'too general' or 'too specific'. We have paid a great deal of attention to these problems. We can determine which contexts should be used in the process and how the new predicates should be chosen. If we did not use the techniques which we have described before our system would not have been able to deal with our set of problems which we have shown before.

(*) See Biermann's paper, page 41.

6.3 WATERMAN: ADAPTIVE PRODUCTION SYSTEMS

In the following we shall give a brief description of Waterman's work. Our main aim will be to describe how his system deals with letter series completion tasks which our system can also deal with.

Letter series completion has been studied by others (Simon & Kotovsky, 1963). We chose to compare our work to Waterman's, since it is more recent (1970) and more closely related to ours.

Production Systems

All knowledge that Waterman's system has about the 'world' is represented in the form of 'production systems' (Newell, 1973). A production system is a collection of 'production rules' of the form

Conditions => Actions.

The left side of each rule contains a set of conditions relevant to the data base referred to as working memory, and the right side contains the list of actions that are supposed to modify the working memory.

Each set of production rules is ordered. The control cycle consists of selecting one production rule from this set and executing the actions. The first rule in the ordered set whose conditions 'match' the working memory is selected. After the actions have been executed the cycle repeats.

A rule to deposit 'C' and 'D' into a working memory if it already

contains 'A' and 'B' is as follows:

(A) (B) => (DEP C) (DEP D)

The set of conditions are written with implicit MEMBER and AND functions. If a rule contains several conditions then all the conditions must match. Moreover the i-th condition must match the i-th item in the working memory because all the items in the working memory are assumed to be ordered.

How New Rules are Obtained

Certain production rules are capable of adding new rules to the existing set of rules. New rules are created from various elements in the working memory. Some elements are used 'conditions' of the new rule and others as 'actions'. More details about how new rules are formed can be found in Waterman's paper.

Production Systems for Letter Series Completion

In the following we shall describe how Waterman's system learns to predict the next letter in the given letter series.

Learning to predict how the given series continues is done in several steps. In each step a partial series is extracted from the given series and an attempt is made to predict the next letter. If the prediction is incorrect the existing system of production rules is modified. When all subseries have been processed the production rules acquired are used to predict the next letter in the series.

For the subseries ABMC, for example, one rule would be generated meaning "if the last three letters of the partial series are ABM then the next letter is C". This is what the rule would look like:

```
(R1)                A   B   M    =>   C
```

Before each new rule is added to the existing set an attempt is made to generalize it taking into account various relationships among the letters. The problem is that for even a relatively simple rule a number of valid generalizations may be found. All the rules shown below, for example, can be obtained by generalization rule R1 shown before.

```
(R2)                x1  B   M    =>   C
(R3)                A   x1  M    =>   C
(R4)                A   B   x1   =>   C

(R5)                x1  x1' M    =>   C
(R6)                A   x1  M    =>   x1'
(R7)                x1  B   M    =>   x1"
```

etc., where

x1' ... successor of x1 (the next letter after x1)
x1" ... successor of x1' (the next letter after x1').

Rule R2, for example, can be interpreted as follows: " If any letter is followed by BM then the next letter is C ". Rule R5 can be interpreted this way: "If any letter is followed by the successor of that letter and by letter M, then the next letter is C ".

If for every new rule the system arbitrarily picked a generalization intending to backtrack whenever necessary a huge tree of possibilities would have been generated making the problem virtually unsolvable. Waterman overcomes this problem by employing a template heuristic which will be described in the following.

Waterman noticed that the letter series dealt with can be split up into several subseries of a fixed length. For example, the series ABMCDMEF can be split up into the following subseries:

ABM, CDM and EF.

Template heuristic consists of hypothesizing the size of the subseries (period size) and recognizing those relations which occupy the same relative position within the subseries.

If, for example, the system was dealing with the subseries ABMC and the period size was 3, only one rule would have been produced by the system:

x1 x2 x3 => x1"

We see that if we employ the 'template heuristic' the number of rules that are generated on the basis of one subseries is substantially reduced.

The period size is assumed to be 1 initially, but it is increased by 1, if no relation is found between the letters occupying the same relative positions in the subseries, or whenever the number of rules added exceeds the current period size.

Waterman has given all the 15 series used by Simon and Kotowsky in 1963 and correct predictions were made in all the cases.

How Our System Differs from Waterman's

We notice that our system is, in certain respects, similar to Waterman's. Knowledge, for example, is in both systems represented in the form of rules (clauses) which the system can modify after an error has been encountered. Our method of generating new rules (clauses) is more sophisticated than Waterman's.

We have mentioned before that in Waterman's system each new rule is generalized before it is added to the existing set of rules. The 'template heuristic' which has been described before restricts the number of rules that the system can generate. Unfortunately, however, the 'template heuristic' a special purpose heuristic and it is useful only when dealing with simple letter series that can be split up in a meaningful way into a number of subseries of a fixed length. (The series 'AMAAMAAAM', for example, cannot be split up this way (*)). The template heuristic is really of no use when we are dealing with problems in other domains (eg. equation solving).

Let us now see what we have done to overcome the problem. We were not really worried about how many new rules our system would generate but rather how many new disjuncts the system would add to the clause dealt with. This is because in our system different alternatives are represented explicitly as disjuncts in the expression generated.

(*) Hedrick (1976) has written a program that can deal with such series.

Let us assume that a particular error has been corrected by adding the expression

D1 v D2 .. Dn

to the clause dealt with. Suppose that we know how the error should have been corrected and let D1 represent that alternative. Clearly, the more alternatives the system has to consider the more likely it is to experience problems. Let us now see how we limit the number of alternatives to be considered.

When our system is dealing with a particular subseries and trying to generate a new clause the other subseries are not always ignored. Obviously, the subseries currently dealt with provides most of the useful information about what the new clause should look like. However, the other subseries are useful, too. This is because sometimes we may want one and the same clause to be used later (with different subseries). Also, sometimes there may be other clauses that should take precedence. Our system takes all this into account and because of that many different alternatives are filtered out.

For example, when our system is dealing with the subseries ABMC, the subseries AB and ABM are also taken into account. The new clause is generated so that the right letter would be predicted in all these cases. Moreover, the clause obtained is simplified when the clause obtained is used with the subseries ABMCDME. This is the clause that has been obtained by our system in the end:

```
series(X1) <-
(X1=:(((..:L3):L2):..) & next(L3,L2)  v X1=:(..:m)) &          pred1
  X1=:    ((..:L2):..) & next(L2,X2)  & ! & write(X2)          pred2
```

We see that the clause contains two kinds of predicates. Some restrict

selection of this clause (pred1) and others determine which letter should be written out as the next letter (pred2).

The predicates constraining selection of this clause have the following meaning: "This clause can be selected if the last letter in the subseries X1 is M, or if the second letter before the end is the successor of the letter appearing in the third position before the end".

The remaining predicates in the clause have the following meaning: "The subseries X1 should continue with the successor of the letter which is in the second position before the end".

If we express the meaning of our clause in the form of production rules we obtain this:

```
(R1)              x1  x2   M   =>  x2'
(R2)         x0   x1  x1'  x3  =>  x1".
```

We notice that we have, in effect, obtained two rules which give us the same answer (letter C) if the series AMB is given to it. The selection conditions of each rule are different. Even though several subseries were used in the process of generating the clause shown before the information considered was not sufficient to reduce the number of conditions to one.

Waterman would in this situation obtain only one rule, thanks to the 'period size hypothesis'. The rule obtained by Waterman is shown the following. (We notice that it is similar to rule R2 shown before.)

```
x1  x2  x3   =>  x1"
```

Even though we do not always end up with the minimal number of rules needed for the given task our method of reducing the number of alternatives to be considered is quite general. It does not matter if we are dealing with algebraic expressions or examples of letter series.

Our system does not always have to generate clauses to correct errors. Certain errors can be corrected simply by reordering the existing set of clauses. That is our system is capable of detecting whether certain ordering of clauses lead to the desired solution of some problem. It can utilize this information later, when it is trying to solve new problems.

Waterman's system does not attempt to correct the errors by reordering the existing rules. This is why his system may, under certain circumstances, generate new rules quite unnecessarily.

7 SUMMARY AND CONCLUSIONS

7.1 PROBLEMS OF EXECUTION CONTROL

Many programs which are written nowadays are so complex that it is often difficult to predict whether some new extension which we want to make could have undesirable side-effects. Many people would agree that if the programs were developed in a systematic manner then they would be not only easier to understand, but also many errors would have been avoided. The question is how do we design programs in this way.

Kowalski (1979) has suggested that we should separate the 'logic' of the program from the 'control' information and perform the design in two separate stages. First, we should design the logic of the program so that the program would give us results which are correct. After we have completed this stage we should see how the results are obtained and try to make the program more efficient (*). That is we should consider how to control the execution. But how can we control the execution?

We believe that priority orderings and predicate

(*) The problem of transforming correct and inefficient programs into more efficient ones has been studied by others. Burstall and Darlington (1977) have developed techniques for transforming programs consisting of 'recursion equations' and Clark and Darlington (1978) have adopted these techniques for 'logic programming'.

constraints which we use in our system provide the programmer with quite a convenient means of control over the execution. The programmer can specify in a simple manner which clause is to be used when.

Obviously, the priority orderings and constraints can be used to control execution of programs consisting of clauses, similar to the ones we use. Also, there is no guarantee that this method of control will helps us to obtain the desired results, if the logic of our program is incorrect.

Role of Priority Orderings and Constraints

Both the priority orderings and predicate constraints that we use are capable of capturing our intuitive knowledge in quite a natural way. Let us consider an example.

Suppose that we want the system to acquire the following rules for adding several integers which should make the system add the integers from left to right.

1. If you are dealing with a term 'X1+X2' see what the left subterm is and if it is not an integer, deal with it first.

2. If you are dealing with a term 'X1+X2' see what the right subterm is, and if it is not an integer, use the rule of associativity to rearrange the subterms by transforming X1+(X2+X3) into (X1+X2)+X3.

If we were to calculate, for example, (1+2)+(2+1) the first rule mentioned tells us that we should calculate 1+2 first. After we have obtained 3+(2+1) the second rule tells us that we should rearrange the subterms so as to obtain (3+2)+1.

All this can be expressed in our language very easily:

```
subs1:  X1+X2=X3       <- X1=:(X5+X6) & ! & X1=X4 & X4+X2=X3
asoc:   X1+(X2+X3)=X4  <-                ! & (X1+X2)+X3=X4
```

Clause 'subs1' is a constrained version of clause 'subs'. It was generated by the system automatically and it was given priority over clause 'asoc'.

Most of the information acquired by the system can be explained in this manner. There is quite a close correspondence between what we know about equation solving, for example, and what the system has acquired. Clause 'over2', for example,

```
over2:  X1+X2=X3 <-  var(X1) & int(X2)  & !  & X1=X3-X2
```

whose constraints have been generated by the system is capable of isolating the variable 'X1' - by transferring the integer 'X2' to the other side of the equation. Isolation of variables is an important technique used in equation solving (Bundy, 1975).

Clause 'asod1' which has also been acquired by the system is also useful when we want to isolate variables. This is how the clause 'asod1' has been constrained:

```
asod1:  (X1+X2)+X3=X4 <-  var(X1) v ... & !  & X1+(X2+X3)=X4.
```

We see that if we were dealing, for example, with the equation

'(X1+3)+1=7', and if we used clause 'asod1' in the first step, we would obtain this equation next: 'X1+(3+1)=7'. The second equation is somewhat easier to solve than the first one, because the variable 'X1' can be isolated in one step.

Equations can often be solved in several different ways and the equation '(X1+3)+1=7' shown before is no exception. If one of the integers is transferred to the other side of the equation, this equation will be obtained: 'X1+3=7-1'. This equation can also be easily solved, because the variable 'X1' can be isolated in one step.

We have tried to be consistent in our experiments, and tried to use the same method of solving equations throughout. In a way we have made it easier for our system to acquire all the knowledge needed for solving a particular set of problems. Further work is needed to establish what the system should do if different ways of solving problems were presented to the system.

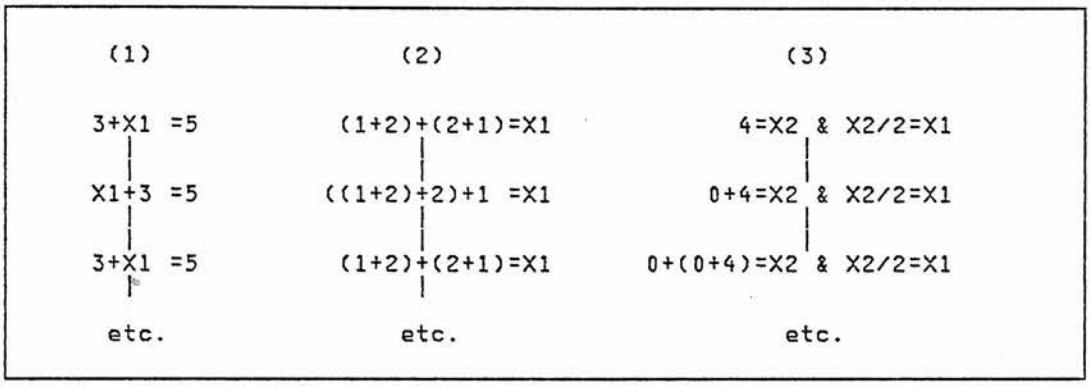
What the system can learn depends to a large extent on what concepts the system is familiar with. We can see that if the system was familiar with the concept of a term containing the unknown and various other rather general concepts, the rules generated by the system would have been more general. The rules obtained would be more similar to the rules which people seem to follow when they are solving equations.

The system was able to learn to solve various equations given quite easily because it was given a number of clauses initially. The 'logic part' of the initial program was 'correct': the clauses given represent various axioms of arithmetic and algebra (and some derived theorems).

The system could have, at least in theory, reached the solution without any additional control information. What the system has acquired is the ability to follow the right branch

in a large search space.

The information acquired enabled our system to avoid many potential 'loops' (*). Several loops which have been eliminated by our system are shown in the following figure.



The first loop arises because the subterms of '3+X1' are repeatedly swapped around as a result of applying the commutativity rule. The second loop arises as a result of using the associativity rule. The third loop arises because '0' is repeatedly added to the left hand side of the equation dealt with.

Limitations of Our Language

Our language is not without limitations, however. Certain statements in English seem to have quite a clear meaning and yet it is quite difficult to express the same thing with clauses and priority orderings.

For example, we cannot specify that some particular clause should be used preferentially before any other clause unless we specify which clauses we mean. This is because the priority

(*) Loops arise whenever the expression obtained is identical (or similar) to the expression handled before.

orderings we use cannot contain variables or any other logical expressions from which we can deduce the name of the clause. This is why the following statement, too, cannot be easily expressed in our language:

"Use rule R preferentially before any other rule dealing with addition".

We believe that despite these limitations our language provides us with simple and useful means of controlling execution of programs.

Developing the Logic of Programs

At the beginning of this chapter we have mentioned that it is best if we design the logic of programs before we consider the questions of efficiency and control. If we can design the logic of programs first then this approach seem to be the right one. However, what do we do, if the logic of our program has not been designed yet? What do we do, if it is incorrect or incomplete?

We have tried to find answers to these questions. We have investigated how various clauses can be obtained from a relatively small number of clauses given to the system. New clauses are obtained from the existing clauses by the addition of new variable instantiating predicates.

The object of these new predicates is to ensure that various variables are instantiated so that the correct answer would be given. A variable is usually not acceptable as the answer.

Suppose that our goal was to add, say, '3' and '2' and the answer 'X' was given. If the answer cannot not be obtained by applying the existing

clauses new clause(s) must be generated. The variable instantiating predicates added must ensure that the correct answer is given.

The given goal trace helps the system to establish how the expression dealt with should be transformed in each step. If goals are not instantiated as they should be the system will try to make changes. It will try to find new predicates that will achieve the desired instantiation(s), and modify the existing clause(s) (see chapter 4).

The method described here was applied mainly to letter series completion tasks. The system learned to predict the next letter correctly. However, the method described is not domain specific. Our work should be continued to see how the system would deal with, say, simple arithmetic, if it had very few clauses available initially. We should investigate how difficult it would be for the system to solve these problems.

7.2 MODIFICATIONS WITHOUT SIDE-EFFECTS

The main objective of our system is detect and correct errors. The aim is to modify the existing set of clauses to eliminate the error detected and all errors of a similar type. However, the solutions of old problems should not be affected by any of the modifications performed. That is the modifications should be without side-effects and we have taken a great care to minimize them.

Explicit priority orderings play a significant role in this. They enable us to detect whether we are trying to rearrange two or more clauses back into the same order as they were before. This is never allowed to happen.

If we find it necessary to give a certain clause a higher priority than before the clause is modified at the same time so that selection of other clauses would not be affected by this change.

The selection contexts of various clauses are identified and used in the process. We have realized that if we want the system to perform certain modifications without side-effects we have to make sure that the system has the knowledge of what should or should not happen in various contexts. Also, the system must know how to identify these contexts. This is what out system can do very well.

It would be wrong to assume that better results will automatically be achieved whenever a large number of contexts is taken into account. We have found that if we want to get the best results the system must be selective and identify only those contexts that actually matter. There is no point in trying to consider other contexts as well. This could actually cause new problems.

Let us imagine that we are treating a patient and that we know that the drug we intend to use will have undesirable side-effects. Obviously it is good to consider how to counteract the effects of that drug, but is no good giving the patient a drug against fictitious side-effects, since this could be harmful.

None of the papers published which we have examined was concerned with this very important problem, that is how use the information available selectively so that the modifications that we want to make would have minimal side-effects.

7.3 CHOICE OF ALTERNATIVES

In any learning system choices may arise as to how the existing program could be modified. How do we decide which of these modifications is the right one?

Suppose we use the following method. We pick one of the alternatives and assume it is the correct one. Then if we get an indication that we were wrong we use some simple backtracking scheme and try another alternative. This method works if we have made only very few choices. The trouble with this method is that as the number of choices gets larger the search space becomes quite unmanageable. Winston has been critical of this method even though he employed it in his system (Winston, 1970). How can we overcome the problem?

We have overcome the problem by employing explicit representation for the set of alternatives to be considered and by trying to eliminate various alternatives as soon as possible.

We have mentioned before that if some clause can be modified by the addition of different predicates then a disjunctive expression is generated containing all of the alternatives found (*).

As each new clause is used in new situations a check is made to see whether any of the predicates added before could be deleted. The system will try to delete those predicates which have a wrong truth-value in the selection context encountered. We have assumed that if the predicate is 'false' then the predicate is not general enough and this is why the

---

(*) If the system can detect that a certain predicate is more specific than another it is not included in this disjunction.

predicate may be deleted. We do not really want to keep the predicates which are not general enough (that is which are 'true' only in some particular context) if we can use better ones instead. This strategy helped us to eliminate various 'odd' alternatives which were introduced by the system initially on the basis of a limited amount of information.

The existing disjunctions of constraints may be modified for other reasons as well. If an error has been detected and the clause to be constrained contains various disjuncts, the system will try to simplify these disjunctions first before trying to correct the error in any other way.

We believe that because we use explicit representation for the set of alternatives to be considered and because we utilize new information as soon as it becomes available our system does not have many difficulties in producing the right modification(s) in the end.

7.4 PREVENTING RECURRENCE OF ERRORS

One popular saying reminds us that we should learn from our mistakes. This bit of popular wisdom is applicable not only to humans, but to systems as well. Both should avoid trying to correct a similar type of error in a similar way as before if this not likely to lead to success.

Why do errors recur? There are many specific reasons why this should happen. However, all different reasons seem to fall into one of the following categories:

1. The system could not identify the cause of the error.

2. The system identified the cause of the error but excluded some important information as irrelevant.

3. The system was incapable of producing the correct modification.

4. The system considered the correct modification but excluded it in preference to another one.

The reasons given above suggest that some errors are more serious than others. Certain types of errors can never be eliminated by the system, because the system is either incapable of seeing what was wrong, or because it is incapable of doing anything about it. This is why these errors will keep recurring.

Our system, for example, is incapable of correcting certain types of errors, because it is incapable of remembering which operation was performed when. Sometimes this is important. (See the discussion on Biermann's work in chapter 6).

Fortunately, there are many errors that can recur but they can be eliminated nevertheless, if the right action is taken.

We have mentioned before that if some error can be corrected by simplification of the existing constraints then other ways of correcting it are not considered at that time. Because of that the correct modification can sometimes be missed out and this is why the error that we wanted to eliminate may recur.

Our system tries to avoid making the same mistake twice. First, it tries to decide whether the modification that it is intending to make is similar to another one performed before. If it is, then the circumstances under which both errors arose

are examined so that the right modification could be produced this time. The additional information helps the system to make the right decision.

The problem of recurrence of errors has not been given much attention in the past. Perhaps, many people believed that if errors recur it is difficult to do anything about it. We have shown that certain errors may be prevented from recurring if one tries.

7.5 INTERACTIVITY

It is clear that no system could learn from experience unless it was given a set of problems to work on and some way of assessing whether that what the system did was right. Our system monitors the execution and tries to decide whether each step performed is right. This is not too difficult because it is given quite a detailed trace showing how each problem should be solved. There is a price to be paid, however. The more details we have to present the more awkward it is for us.

We should investigate how we could avoid giving the system all the details. Often the details are not really needed. The system may already know how to solve a particular set of subproblems used in the solution of the given problem.

Suppose we are trying to show the system how simple equations should be solved and at some stage we need to do some arithmetic. The system does not need to be shown how to add or subtract numbers if it already knows that.

What we should do is try to help the system just where we

think it could have difficulties. This is more difficult than it sounds. It is not easy to guess what the system can or cannot do unless we have an exact model of the system that we are trying to teach. What we need is a better ability to communicate with the system. More work is needed to establish how this can be achieved.

7.6 APPLICATIONS

Automatic programming is a relatively young discipline. It will take some time before all the issues involved are well understood so that we will be able to develop rather complex programs with the help of systems like ours.

However, several learning programs which have already been written in the past have proved themselves rather successful. Buchanan, Mitchell and others (1977) have written a program DENDRAL which can identify an unknown compound on the basis of mass spectroscopy. The system can formulate hypotheses about the molecular structure of the given compound on the basis of the given specrograph. The hypotheses are produced by the set of rules given to the system.

Meta-DENDRAL is an extension of DENDRAL. It is capable of formulating new rules on the basis of experimental data. It inspects the experimental data to see if there are any conspicuous patterns in it and then it tries to generate new rules which would produce these patterns.

Several other systems which have been developed are now widely used in practice. Some can be trained to recognize signatures; others can be trained to recognize sounds. So far,

each system was rather a special purpose system. Our work shows that one and the same system could be trained to do a number of very different tasks. Also, our work helps us to obtain a better understanding of the complex issues involved, and so it should be easier to build similar systems in future.

REFERENCES

Biermann A W (1977): The Inference of Regular LISP Programs from Examples, Dept. of Computer Science, Duke University

Brazdil P (1977): Learning Simple Arithmetic, Proc. of 5th IJCAI, Cambridge, Mass.

Brazdil P (1978): Experimental Learning Model, Proceedings of AISB Summer Conference, Hamburg

Buchanan B G, Mitchell T M (1977): Model-Directed Learning of Production Rules, Stanford Heuristic Programming Project Memo HPP-77-6, Stanford University

Bundy A (1975): Analysing Mathematical Proofs, DAI Research Report No.2, Edinburgh

Burstall R M, Darlington J (1975): Some Transformations for Developing Recursive Programs, Proc. of 1975 Int. Conference on Reliable Software, Los Angeles

Clark K L, Darlington J (1980): Algorithm Classification through Synthesis, The Computer Journal, Vol 23, No 1, pp 61-65

Fikes R, Hart P, Nilsson N (1972): Learning and Executing Generalized Robot Plans, Artificial Intelligence Vol. 3, pp. 251-288

Hardy S (1975): Synthesis of LISP Functions from Examples, Advance Papers of 4th IJCAI, Tbilisi, USSR, pp. 240-245

Hedrick C L (1976): Learning Production Systems from Examples, Artificial Intelligence, Vol.7, No.1

Jouannaud J P, Kodratoff Y (1978): Quelques Method Analytique de Syntese Automatic de Programmes a Partir d'Examples, Institut de Programmation, Universite de Paris VI

Kowalski R A (1979): Logic for Problem Solving, Artificial Intelligence Series, North Holland Publishing Co., Amsterdam

McCarthy J, Abrahams P W, Edwards D J, Hart T P, Levin M I (1962): Lisp 1.5 Programmer's Manual, MIT Press, Cambridge, Mass.

Newell A (1970): Production Systems: Models of Control Structures, Visual Information Processing, Chase W (Ed.), Academic Press

Plotkin G D (1969): A Note on Inductive Generalization, Machine Intelligence 5, Edinburgh University Press, Edinburgh

Simon H A, Kotowsky K (1963): Human Acquisition of Concepts for Sequential Patterns, Psychological Review, Vol.70, pp 534-546

Soloway E M, Riseman E M (1976): Mechanizing the Common Sense Inference of Rules which Direct Behaviour, Proceedings of AISB Summer Conference, University of Edinburgh

Smith R G, Mitchel T M, Chestek R A, Buchanan B G (1977): A Model for Learning System, Report No. Stan-CS-77-605

Sussman G J (1975): A Computer Model of Skill Acquisition, New York: American Elsevier

Vere S A (1977): Induction of Relational Productions in the Presence of Background Information, Proc. of 5th IJCAI, Cambridge, Mass., pp. 349-355

Warren H D (1977): Implementing Prolog, Research Report 39 & 40, Dept. of AI, University of Edinburgh

Waterman D A (1970): Adaptive Production Systems, Complex Information Processing Paper 285, CMU Dept. of Psychology

Winston P H (1970): Learning Structural Descriptions from Examples, Ph.D. Thesis MIT AI-TR-231

APPENDIX - GLOSSARY

clause

A clause is an expression of the form H <- B, where 'H' is a 'clause head', and 'B' is a 'clause body'.

clause head

A clause head is a 'predicate'. In our extended system ELM2 a clause head is an expression of the form 'P1 & G1', where 'P1' is a 'predicate' and 'G1' is a variable.

clause body

A clause body is an expression of one of the following forms:

```
Cs & ! & Pn    or
     ! & Pn    or
Cs & !         or
     !,
```

where Cs represents 'constraints' and 'Pn' represents a 'predicate', or a conjunction of 'predicates'.

clause instance

A 'clause instance' of some clause C is a 'substitution instance' of that clause. It is obtained from clause C by substituting one or more variables in it by 'terms'. All variables with the same name have to be replaced by the same term.

constraint

A constraint is a predicate, or a disjunction of constraints, or a conjunction of constraints. The truth or falsity of constraints is established during 'clause selection'.

context

A context is a conjunction or a disjunction of 'goals'.

goal

A goal is a 'predicate', or a conjunction of 'goals', or a disjunction of 'goals', whose truth or falsity is to be established.

matching

Term T1 matches term T2, if T1 and T2 can be made identical by substituting the existing variables in T1 or T2 by 'terms'. All variables occurring in T1 (or T2) which have the same name have to be replaced by the same term.

meta-predicate

A 'predicate' whose truth or falsity is determined by the system without regard to any clauses provided by the user.

predicate

A predicate is an expression of the form P(T1,..Tn), where P is a predicate name and T1,..Tn are 'terms'.

selection context

A selection context of clause C is the 'context' in which this clause has been selected, or in which it should be selected

rejection context

A rejection context of some clause is the 'context' in which this clause should be rejected from selection.

term

A term is a variable, a constant or of the form F(T1,..Tn), where 'F' is an n-adic function name and T1,..Tn are terms.

PUBLISHED PAPERS

EXPERIMENTAL LEARNING MODEL

Pavel Brazdil

Department of Artificial Intelligence
University of Edinburgh, Scotland

ABSTRACT: The aim of our work is to investigate how a relatively small set of rules can be transformed into a running program capable of solving a fairly wide variety of problems. We have written a program ELM (Experimental Learning Model) which modifies a given set of rules (clauses) as a result of its experience. The problems are chosen from the domain of school arithmetic/algebra and solved by one part of our system (with the help of the teacher). The solutions are analyzed and modifications to the existing system of rules are performed as a result.

CONTENTS:

1. Input Provided by the Teacher
2. Search for a Solution
3. Assimilation of Priority Orderings
4. Conflict Resolution
5. Results and Conclusions

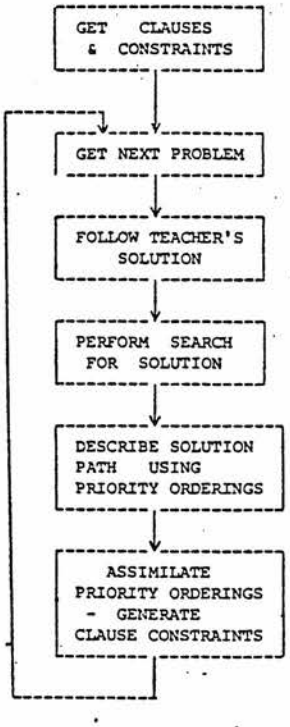


FIG.1 BASIC VIEW OF ELM

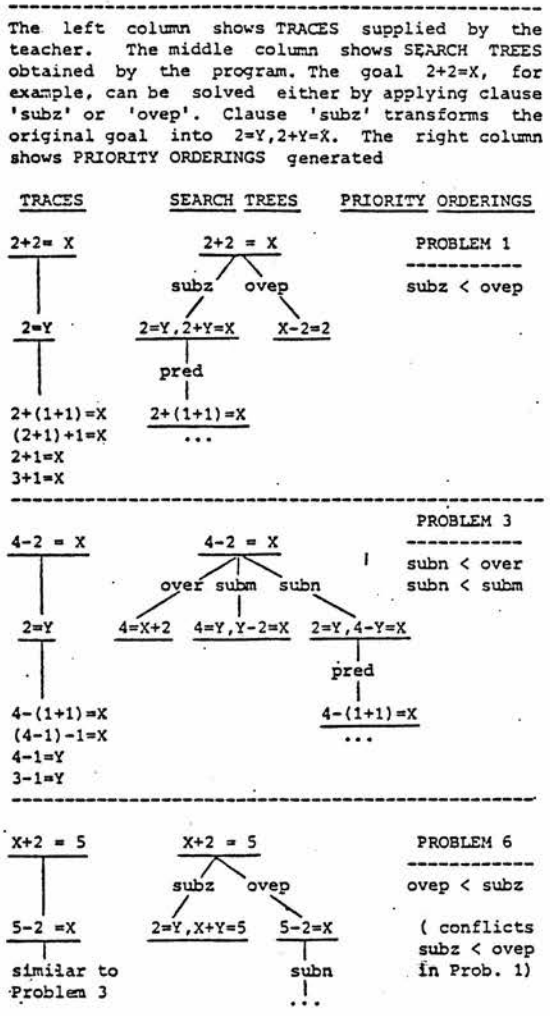
1. INPUT PROVIDED BY THE TEACHER

In this section we will concentrate on the description of the information the learning model is provided by the teacher. Later, we will show how problems are solved and what kinds of modifications our system can produce. Fig.1 shows the basic view of the system.

(1) At present ELM needs to be provided with a set of clauses which express various rules the teacher might consider useful for solving a certain class of problems. Fig.2 shows such a set of clauses intended for manipulating arithmetic and algebraic expressions. For example, the clause 'asoc' expresses associativity. The clauses contain little information about when they should be used. We will use our work to show that it is relatively easy to generate this information automatically.

(2) We provide the system with a set of predicates (constraints), which the system may insert among the existing predicates in the clause. Their function will be to constrain clause selection. In the implementation we have used the following:

predicate	var(X)	true if	X is a variable,
	int(X)	-"-	X is an integer,
	X:=U+V	-"-	X is of the form U+V.



Experience with Prob.3 facilitates the solution of Problem 6, since the subgoal 5-2=X is solved without search. Problem 1, on the other hand, eliminates temporarily the branch leading to the the solution of X+2=5. Thanks to trace provided the error is soon discovered and rectified.

FIG.4 EXAMPLES OF TRACES AND SEARCH TREES

```
asoc:  X1+(X2+X3)=X4 <- (X1+X2)+X3=X4.
ovep:  X1+X2=X3 <- X3-X2=X1.
over:  X1-X2=X3 <- X1=X3+X2.
subm:  X1-X2=X3 <- X1=X0 & X0-X2=X3.
subn:  X1-X2=X3 <- X2=X0 & X1-X0=X3.
subs:  X1+X2=X3 <- X1=X0 & X0+X2=X3.
subz:  X1+X2=X3 <- X2=X0 & X1+X0=X3.
pred:  1=1+0 <-.    2=1+1 <-.    etc.
suc:   0+1=1 <-.    1+1=2 <-.    etc.
```

FIG.2 EXAMPLE OF CLAUSES GIVEN TO ELM

(3) ELM needs to be given a sequence of problems to solve. Fig.3 shows an example of problems given to ELM. Each problem is solved by evaluating the arithmetic expressions and/or by finding the correct value for the variable in the expression.

(4) Generally, learning models may require some form of search guidance in their search for solution. ELM is shown a sequence of subgoals as they are to be solved - a trace. An example of a trace is shown in Fig.4. Traces are produced by a routine which in effect simulates the teacher.

2. SEARCH FOR A SOLUTION

ELM responds to each problem by initiating a search for a solution. For each subgoal a set of clauses is selected, provided they match. The ordering of clauses is also respected. Each clause, when applied replaces the current subgoal by its body (Fig.4). The application creates a new node in the search tree, which is expanded breadth-first. The trace provided is utilized to terminate a branch whenever the current subgoal differs from the one in the trace. The search terminates whenever there are no more subgoals to be solved,

```
1/  2+2 =X
2/  (1+1) +2 =X
3/  4-2 =X
4/  (5-1) -2 =X
5/  5-3 =X
6/  X+2 =5
7/  X+ (1+1) =5
8/  ((1+X) +2)+3 =8
9/  X+3=Y &  Y+1=9
```

FIG.3 A TYPICAL SEQUENCE OF PROBLEMS GIVEN

or whenever no clause can be found for solving a particular subgoal. The answer 'yes', or 'no' respectively, is given.

The search trees obtained by the problem solving routine are analyzed with the objective to eliminate search. Experience with one problem often facilitates solution of similar problems in the future, but difficulties can also be introduced. Illustrations of that are given in Fig.4. Thanks to the trace shown the errors are soon discovered and rectified by modification of clauses. The basic strategy of ELM is to restrict selection criteria for clauses that do not lie on the solution path Either priority orderings or clause constraints can be introduced.

(1) Priority orderings specify the order in which the clause selection should be performed. The priority ordering C<D, for example, indicates that the clause C should be selected preferentially before D. The selection of clause D is restricted as a result. The priority orderings are generated as as the solution path in the search tree is described *.

(2) Clause constraints can be used as an alternative means for restricting clause selection. Clause constraints are predicates whose truth/falsity is tested during clause selection. If they are not true the clause to be selected is ignored.

3. ASSIMILATION OF PRIORITY ORDERINGS

After the solution of each given problem has been reached the priority orderings describing the solution path are assimilated with the existing ones. They are added to the orderings previously remembered unless conflicts are detected, in which case new clauses are generated.

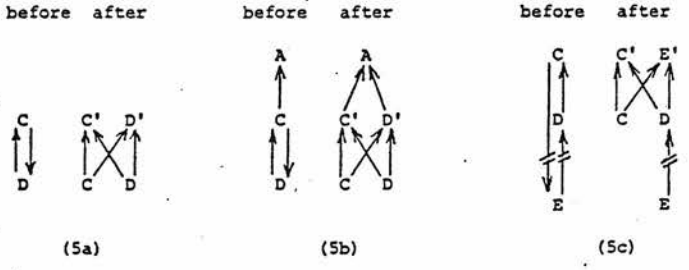


FIG. 5 PRIORITY ORDERINGS
(before and after conflict resolution)

For example, the conflict between C<D and D<C is resolved by generating two new clauses C' and D', constrained versions of C and D. They constrained versions are always given preference to the unconstrained ones (Fig.5a). We prefer to generate both clauses C' and D' although in principle we could have generated just one of them. However, we would have to make a choice, which we prefer to avoid.

---

* Sussman (1973) discusses the role of a 'critic' in his learning system HACKER. The description of the solution solution path can be viewed as 'criticism' of the present clause selection process.

The old clauses C and D are kept as a back-up and used whenever the new clauses do not get selected. Conflicts involving three or more clauses are resolved in the way outlined in Fig.5c. Some additional priority orderings are sometimes generated as a result of integration of the new clauses C' and D' among the existing clauses. Two such priority orderings, shown in Fig. 6b, ensure that the priority given to clause A is unaffected by conflict resolution.

4. CONFLICT RESOLUTION

The conflict resolution routine requires the contexts in which the clauses to be constrained were selected or rejected. The subgoal solved when clause C was selected (eg. when C<D was generated) is used as its selection context. The rejection context is defined similarly. Both contexts are stored together with the priority orderings generated. Had they not been stored, appropriate problem(s) would have to be found, the solutions re-created and the desired contexts retrieved. The conflict resolution routine takes each clause to be constrained and searches for its variables and corresponding terms in the selection and rejection contexts. Suppose that the routine is trying to resolve the conflict between subz<ovep and ovep<subz (Fig.4), and is just considering the following terms:

Clause head:	Selection context:	Rejection context:
X1+X2=X3	2+2=X	X+2=5
'	'	'

The conflict resolution routine takes each possible constraint (var(X), int(X) or X:=T) and applies it to the terms chosen. If the predicate is true when applied to the selection context and, at the same time, false when applied to the rejection context it is added as a disjunct to the expression being built. Since for example int(2) is true and int(X) false, the predicate int(X1) is added to the existing constraints. Further investigation yields var(X3), which is added as a disjunct to the constraint int(X1) found previously.

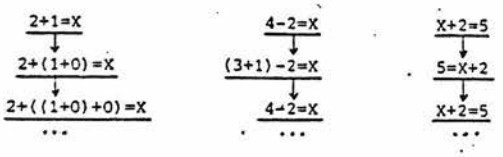
Disjunctions of constraints are preferred because arbitrary choice of a constraint does not need to be made. Also, we prefer more general constraints to special ones in order to avoid producing overspecialized and generally inapplicable clauses.

For example,	int(X)	is preferred to	X:=2,
	X:=X1+X2	-"-	X:=2+1,
	int(X1) v var(X3)	-"-	int(X1).
Also,	C1 & (C1v C2)	is replaced by	C1,
	(C1v C2) & (C1v C3)	-"-	C1.

The last rule is used quite often by our program. The repeated occurence of the same predicate can be taken as evidence that it is relevant for clause selection while all the others are merely incidental.

5. RESULTS AND CONCLUSIONS

The learning model ELM discussed in this section has been implemented in PROLOG. In one of our experiments the program was given 20 clauses, but no priority orderings. Sequence of problems was also given (Fig.2,3). The search trees contained only one branch leading to the solution after one run through the learning sequence. Although the program knew nothing about 'loops' many were eliminated nevertheless, eg. the ones shown. In total twenty-two new clauses were generated, two for each conflict detected. Some of the clause new constraints are shown in Fig.6b, and corresponding priority orderings in Fig.6a. We are planning to extend the set of possible constraints to include others, which should enable us to deal with more complex equations in the future. Also, we intend to utilise to a greater extent the information in the trace, basically to generate clauses.




```
over.2:  X1-X2=X3 <- (X1:=3  v X2:=1) & int(X1)
over.1:  X1-X2=X3 <- int(X1)     ...
subm.1:  X1-X2=X3 <- X1:=X4-X5   ...
subh.1:  X1-X2=X3 <- X1:=4   v X2:=2      ...

ovep.1:  X1+X2=X3 <- var(X1) v int(X3)    ...
subs.1:  X1+X2=X3 <- X1:=X4+X5 v X2:=1    ...
subz.2:  X1+X2=X3 <- int(X1)     ...
subz.1:  X1+X2=X3 <- int(X1) v X2:=2      ...
```

FIG. 6b EXAMPLES OF CONSTRAINTS GENERATED
(Other predicates follow the constraints shown)

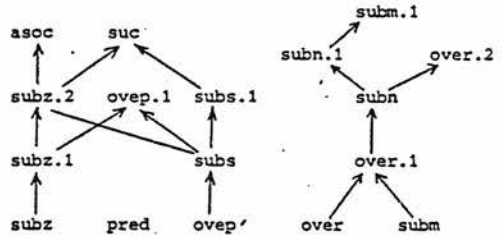


FIG. 6a SOME PRIORITY ORDERINGS GENERATED
(The arrows point to higher priority clauses)

The basic contribution of our work is, we believe, in that we show how analysis of problem solving activity can be used to produce systematic modifications to the existing program.

ACKNOWLEDGEMENTS AND REFERENCES

I wish to thank to Gordon Plotkin for many helpful discussions and comments on this paper. Also, thanks to IBM UK Ltd. for their help and support in the course of my stay in Edinburgh.

Sussman, G.J. (1973) A Computational Model of Skill Acquisition, MIT Tech. Report AI TR-297, MIT, Cambridge

Vere, S.A. (1977) Induction of Relational Productions in the Presence of Background Information, Proceedings of 5th IJCAI, 1977, MIT, Cambridge

Winston, P.H. (1970) Learning Structural Descriptions from Examples, MIT Tech. Report AI TR-231, MIT, Cambridge

Waterman, D.A. (1975) Adaptive Production Systems, Proceedings of 4th IJCAI, 1975, Tbilisi, USSR